THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

High Performance Bioinformatics and Computational Biology on General-Purpose Graphics Processing Units

Cheng Ling

A thesis submitted for the degree of Doctor of Philosophy

The University of Edinburgh

February 2012

Abstract

Bioinformatics and Computational Biology (BCB) is a relatively new multidisciplinary field which brings together many aspects of the fields of biology, computer science, statistics, and engineering. Bioinformatics extracts useful information from biological data and makes these more intuitive and understandable by applying principles of information sciences, while computational biology harnesses computational approaches and technologies to answer biological questions conveniently. Recent years have seen an explosion of the size of biological data at a rate which outpaces the rate of increases in the computational power of mainstream computer technologies, namely general purpose processors (GPPs). The aim of this thesis is to explore the use of off-the-shelf Graphics Processing Unit (GPU) technology in the high performance and efficient implementation of BCB applications in order to meet the demands of biological data increases at affordable cost.

The thesis presents detailed design and implementations of GPU solutions for a number of BCB algorithms in two widely used BCB applications, namely biological sequence alignment and phylogenetic analysis. Biological sequence alignment can be used to determine the potential information about a newly discovered biological sequence from other well-known sequences through similarity comparison. On the other hand, phylogenetic analysis is concerned with the investigation of the evolution and relationships among organisms, and has many uses in the fields of system biology and comparative genomics. In molecular-based phylogenetic analysis, the relationship between species is estimated by inferring the common history of their genes and then phylogenetic trees are constructed to illustrate evolutionary relationships among genes and organisms. However, both biological sequence alignment and phylogenetic analysis are computationally expensive applications as their

computing and memory requirements grow polynomially or even worse with the size of sequence databases.

The thesis firstly presents a multi-threaded parallel design of the Smith-Waterman (SW) algorithm alongside an implementation on NVIDIA GPUs. A novel technique is put forward to solve the restriction on the length of the query sequence in previous GPU-based implementations of the SW algorithm. Based on this implementation, the difference between two main task parallelization approaches (Inter-task and Intra-task parallelization) is presented. The resulting GPU implementation matches the speed of existing GPU implementations while providing more flexibility, i.e. flexible length of sequences in real world applications. It also outperforms an equivalent GPP-based implementation by 15x-20x. After this, the thesis presents the first reported multi-threaded design and GPU implementation of the Gapped BLAST with Two-Hit method algorithm, which is widely used for aligning biological sequences heuristically. This achieved up to 3x speed-up improvements compared to the most optimised GPP implementations.

The thesis then presents a multi-threaded design and GPU implementation of a Neighbor-Joining (NJ)-based method for phylogenetic tree construction and multiple sequence alignment (MSA). This achieves 8x-20x speed up compared to an equivalent GPP implementation based on the widely used ClustalW software. The NJ method however only gives one possible tree which strongly depends on the evolutionary model used. A more advanced method uses maximum likelihood (ML) for scoring phylogenies with Markov Chain Monte Carlo (MCMC)-based Bayesian inference. The latter was the subject of another multi-threaded design and GPU implementation presented in this thesis, which achieved 4x-8x speed up compared to an equivalent GPP implementation based on the widely used MrBayes software.

Finally, the thesis presents a general evaluation of the designs and implementations achieved in this work as a step towards the evaluation of GPU technology in BCB computing, in the context of other computer

technologies including GPPs and Field Programmable Gate Arrays (FPGA) technology.

Declaration of Originality

I hereby declare that the research reported in this thesis and the thesis itself was composed and originated entirely by myself in the School of Engineering at The University of Edinburgh.

Cheng Ling
February 2012
Edinburgh, UK

Acknowledgements

In the first instance, I would like to express my deepest gratitude to my supervisor, Dr. Khaled Benkrid. Under his guidance, I could obtain professional knowledge on this work and finish the thesis. He provided me with valuable suggestions and criticisms, with his profound knowledge. His patience and kindness were greatly appreciated. Besides, he was always attaching great importance to the thesis writing methods and formats.

I would like to thank Dr. Ahmet Erdogan for his help through my study in SLIG. Moreover, colleagues in SLIG office, including Mr. Wei Zhou, Mr Yi Ding, Mr. Chuan Hong, Mr. Haoyu Zhang and all of those who have helped me, are also gratefully acknowledged.

I am also very grateful to my older brother, Dr. Linzhong Xia and Mr. Liwen Ling, thanks for their great encouragement and suggestions.

Finally, I would like to thank my parents for their supports in all forms!

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BCB	Bioinformatics and Computational Biology
BLAST	Basic Local Alignment Search Tool
CLP	Conditional Likelihood Probability
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DNA	Deoxyribonucleic acid
DSP	Digital Signal Processor
DRAM	Dynamic Random Access Memory
FPGA	Field Programmable Gate Array
GPGPU	General-purpose computation Graphics Processing Units
GPP	General-purpose Processor
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
HPC	High Performance Computing
HSP	High Scoring Pair
MCMC	Markov Chain Monte Carlo
MCUPS	Mega Cell Updated Per Second
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
ML	Maximum Likelihood
MP	Maximum Parsimony
MSA	Multiple Sequence Alignment
NJ	Neighbour Joining
NP	Nondeterministic Polynomial time
NRE	Non-Recurring Engineering
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
PGA	Professional Graphic Adapter
RISC	Reduced Instruction Set Computing
RNA	Ribonucleic acid
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
SGI	Silicon Graphics Inc.
SM	Stream Multiprocessor
SP	Stream Processor
SW	Smith-Waterman Algorithm
T&L	Transform and Lighting
UPGMA	Un-weighted Pair Group Method with Arithmetic Mean
VLSI	Very Large Scale Integration

Chapter 1
Introduction

1.1 Introduction and Motivation

Bioinformatics and Computational Biology (BCB) is a relatively new multidisciplinary field which brings together many aspects of the fields of biology, computer science, statistics, and engineering. BCB aims to develop systems that help extract and analyse biological information in a convenient and speedy way. Bioinformatics extracts useful information from biological data and makes these more intuitive and understandable by applying principles of information sciences, while computational biology harnesses computational approaches and technologies to resolve biological questions conveniently [1]. The success in transcribing complete biological sequences, e.g. DNA, RNA and Protein, from biological samples is the main reason behind the emergence and fast growth of BCB, with real world applications in drug engineering, bio-material engineering and disease diagnosis.

The explosion of the size of biological data however poses enormous challenges to current computing technologies. Indeed, the rate at which biological data is growing outpaces the rate of increases in the computational power of mainstream computer technologies, namely General-purpose Processors (GPPs). High performance supercomputers and computer clusters have been proposed as efficacious implementation platforms for high performance BCB applications. However, high cost and lack of suitable programming interfaces restrict a wider employment of these platforms. Moreover, supercomputers are often adorned with special-purpose hardware, e.g. in the form of ASICs [2][3], in order to achieve considerable performance at a relatively low power consumption. Nevertheless, ASICs suffer from high Non-Recurring Engineering (NRE) costs and lack of flexibility as ASICs cannot normally be reused to implement other algorithms. Field Programmable Gate

Arrays (FPGAs) have been proposed as another viable implementation platforms for BCB applications due to their ASIC-like performance with the additional programmability feature. However, FPGAs also suffer from high development and purchase costs compared to general purpose processors [4].

Graphics Processing Units (GPUs) have been proposed as high performance computing platforms and relatively low cost for a variety of general-purpose computing applications. Central to this is the huge economies of scale this technology benefits from through the gaming industry, as well as the development of a high level general computing Application Programming Interface (API) for them, called Compute Unified Device Architecture (CUDA) from NVIDIA corporation. The latter is based on a standard programming language e.g. C, with high level API functions to exploit the architecture parallelism. These API functions have contributed greatly to the wider use of GPUs in general purpose computing, opening the way for a new field of computational study coined general-purpose computation GPU (GPGPU). The aim of this thesis is to explore the use of off-the-shelf GPU technology in the high performance and efficient implementation of BCB applications in order to meet the demands of biological data size increases at affordable cost.

1.1.1 Genomic Data Primer

DNA is the abbreviation of Deoxyribonucleic Acid, which is the most basic genomic data. It is the main chemical components of chromosomes and also the material composing genes in living cell. DNA is referred to as "genetic particles", as parent generations duplicate part of their own DNA and pass it to descendents, thus spreading of traits through generations. DNA has four basic nucleotide structures, which are adenine (A), cytosine(C), guanine (G) and thymine (T). It is composed of two single vine-shape chains of nucleotides, with one wound around another, to form a spiral. According to a different taxonomy, the spiral can be divided into A-type DNA, B-type DNA and Z-type DNA. James Watson and Francis Crick [5] first discovered the B-type DNA

double helix, which is the most common type in cells. A DNA sequence is able to self-replicate, and its physiological function is expressed in the form of proteins. For this, Ribonucleic Acid (RNA) is first generated by DNA through transcription. It is then translated into proteins which are involved in all aspects of cell assembly and functions. RNA's basic nucleotides are: adenine (A), cytosine(C), guanine (G) and uracil (U) instead of thymine. Three RNA nucleotides are translated into one amino acid which is the basic building block of proteins. The basic coding units of RNA (3 nucleotides) are referred to as "codons", some of which carries genetic information and are used to express common amino acid symbols. The total number of 3-letter combinations of coding units is 64, but only 20 amino-acids form proteins. The abbreviation and full name of all the twenty amino acids is listed in Table 1.1.

Table 1.1: Full name and single-letter code of amino acids

Isoleucine(I)	Leucine(L)	Valine(V)	Phenylalanine(F)
Methionine(M)	Cysteine(C)	Alanine(A)	Glycine(G)
Proline(P)	Threonine(T)	Serine (S)	Tyrosine(Y)
Tryptophan (W)	Glutamine(Q)	Asparagine(N)	Histidine(H)
Glutamic acid(E)	Aspartic acid(N)	Lysine (K)	Arginine(R)

Figure 1.1 illustrates the correspondence between the 64 codons and the 20 amino acid symbols. The discovered DNA and Protein sequences are well classified, stored and indexed in genetic databases. Two popular databases are known as SWISS-PROT [6] and GenBank [7]. SWISS-PROT is a protein sequence database, the last release of which contains 532,792 sequence entries. GenBank is an annotated collection of all publicly available DNA sequences, which contains approximately 135 million sequence records. Recent years have seen an explosion in the size of the biological database. Figure 1.2 illustrates that the growth trend of the biological sequence database is exponential over time.

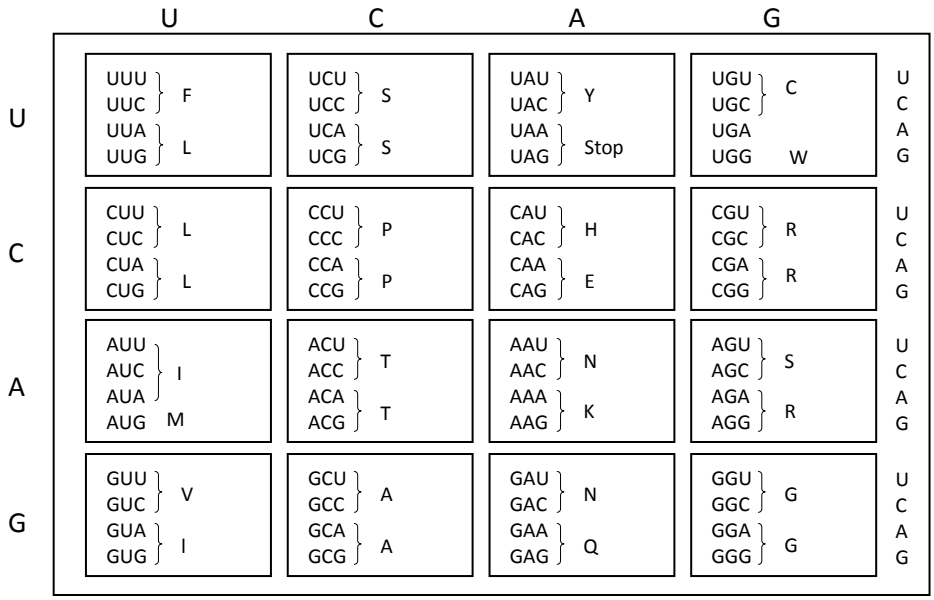


Figure 1.1: All the twenty amino acids and their relationship with nucleotides

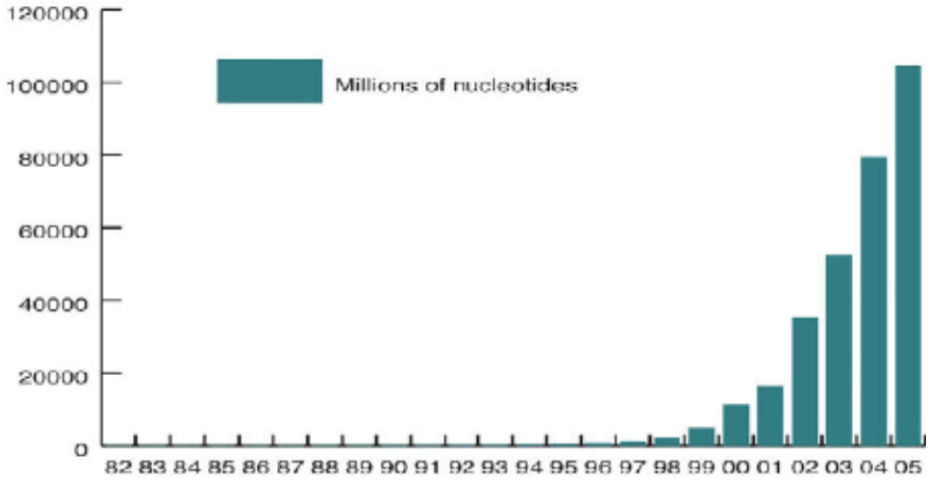


Figure 1.2: Exponential growth of biological sequence database over year [6]

1.2 Thesis objectives and Contributions

The thesis presents the detailed design and implementation of GPU solutions for a number of BCB algorithms in two widely used BCB applications, namely biological sequence alignment and phylogenetic analysis. Biological sequence alignment is widely used operation in bioinformatics and genetics research, which aims to find the best possible alignment between biological sequence

sequences [8]. Biological sequence alignment also can be used to determine potential information about a newly discovered biological sequence from other well-known sequences through similarity comparison. In addition, it can also be useful for the research of biological evolution history. The most basic sequence analysis task is to ask if two sequences are related. This is usually done by first aligning the sequences (or part of them) and then deciding whether that alignment is more likely to have occurred because the sequences are related, or just by chance. Phylogenetic analysis is the investigation of the evolution and relationships among organisms, and is widely used in the fields of system biology and comparative genomics [9]. It is particularly vital in drug and vaccine development [10]. In molecular based phylogenetic analysis, the relationship between species is estimated by inferring the common history of their genes and then phylogenetic trees are constructed to illustrate evolutionary relationships among genes and organisms [11].

However, both biological sequence alignment and phylogenetic analysis are computationally expensive applications as their computing and memory requirements grow heavily with the size of the sequence databases - leading to long execution times on GPPs, hence the need for hardware acceleration of these tasks [8].

The detailed objectives of this thesis are therefore as follows:

- Design and implementation of an efficient biological sequence alignment application on GPUs, using the widely used Smith-Waterman algorithm
- Design and implementation of an efficient biological sequence alignment application on GPUs, using the widely used heuristic BLAST algorithm
- Design and implementation of an efficient GPU solution for multiple sequences alignments

- Design and implementation of an efficient GPU solution for phylogenetic analysis
- Evaluation of GPUs in high performance bioinformatics and computational biology, compared with state-of-the-art computer technologies

The contributions of this thesis are as follow:

- A novel multi-threaded parallel design of the Smith-Waterman (SW) algorithm [12] alongside an implementation on NVIDIA GPUs is achieved. A novel technique is put forward to solve the restriction on the length of the query sequence in previous GPU-based implementations of the SW algorithm. Moreover, we achieved the algorithm by another parallelization approach and evaluated the difference between Inter-task and Intra-task parallelization approaches.

- A multi-threaded design and GPU implementation of the Gapped BLAST with Two-Hit method algorithm [13] is achieved, which is the first GPU-based implementation that ever reported in literatures.

- A multi-threaded design and GPU implementation of a Neighbor-Joining (NJ)-based [14] method for phylogenetic tree construction and multiple sequence alignment is achieved.

- Since the NJ method only gives one possible tree, another multi-threaded design and GPU implementation by the more advanced Markov Chain Monte Carlo (MCMC)-based Bayesian inference method for phylogenetic analysis is achieved.

- A general evaluation of the designs and implementations achieved in this work as a step towards the evaluation of GPU technology in BCB

computing is presented, in the context of other computer technologies including GPPs and Field Programmable Gate Arrays (FPGA) technology.

1.3 Thesis Structure

The remainder of the thesis is organised as follows:

- Chapter 2 presents fundamentals and characteristics of Graphics Processing Units (GPU). Furthermore, the graphics pipelines utilized by traditional GPU architectures and the developments of GPUs for the last 20 years are laid out before the architectural details of the compute unified device architecture (CUDA) GPUs are presented. The thread batching strategy and memory models of the GPU used in our experiments are finally introduced.

- Chapter 3 first introduces essential background information on the widely used Smith-Waterman algorithm (SW). Furthermore, a multi-threaded parallel design and implementation of SW algorithm on CUDA-compatible graphic processing units (GPUs) is presented. A novel technique is put forward to solve the restriction on the length of the query sequence in previous GPP-based implementations of the SW algorithm. Finally, this chapter presents the difference between the Intra-task parallelization strategy and the Inter-task parallelization strategy through the description of task loading balance performance comparisons.

- Chapter 4 initially introduces essential background information on Basic Local Alignment Search Tool (BLAST) [15], which is a very popular heuristic algorithm for biological sequence alignment. Afterwards, the implementation of the Gapped BLAST for biological sequence alignment on CUDA-compatible GPUs, with the Two-Hit method, is presented.

Following this, performance comparisons between the proposed method and the most optimized CPU-based NCBI BLAST, is laid out.

- Chapter 5 presents essential background on the Myers-Miller algorithm [16], which is developed to compute optimal alignments in linear memory space. A multi-threaded design and GPU implementation of a Neighbour-Joining (NJ)-based method for phylogenetic tree construction and multiple sequence alignment (MSA) is then presented. Performance comparisons to ClustalW software [17] are finally laid out.

- Chapter 6 first presents essential background on Phylogenetic analysis and the more advanced Maximum Likelihood (ML) [18] method for scoring phylogenies. A GPU-based multi-threaded design and implementation of the Maximum likelihood method, with MCMC-based Bayesian inference for phylogenetic analysis on a set of aligned amino acid sequences is then presented. Performance comparisons between the proposed GPU-based method and the MrBayes software [19] are finally laid out.

- Chapter 7 presents a general evaluation of the designs and implementations achieved in this work as a step towards the evaluation of GPU technology in BCB computing, in the context of other computer technologies including GPPs and Field Programmable Gate Arrays (FPGAs) technology.

- Chapter 8 presents general conclusions and avenues for future work.

1.4 Published Papers

Ling C. and Benkrid K.: A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs. Proc. of the Application Specific Processors (SASP), 2009, pp.94-100

Ling C. and Benkrid K.: Design and implementation of a CUDA-compatible GPU-based core for gapped BLAST algorithm. Proc. of the International Conference on Computational Science, 2010, pp.495-504.

Ling C., Benkrid K. and Erdogan A.T.: High performance Intra-task parallelization of Multiple Sequence Alignments on CUDA-compatible GPUs. Proc. of the Adaptive Hardware and Systems (AHS), NASA/ESA Conference, 2011, pp.360-366

Khaled Benkrid, Ali Akoglu, Cheng Ling, Yang Song, Xiang Tian and Ying Liu: The Promise and Challenges of High Performance Efficient Reconfigurable Computing, accepted by International Journal of Reconfigurable Computing

Paper under review:

Cheng Ling and Khaled Benkrid: Design and Implementation of High Performance Multiple Sequence Alignments on CUDA – Compatible GPUs, submitted to bioinformatics

1.5 References

[1] Gokhale M. and Graham P.S.: Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays. Springer, 2005.

[2] Shimokawa Y., Fuwa Y. and Aramaki N.: A parallel ASIC VLSI neurocomputer for a large number of neurons and billion connections per second speed. Proc. of the IEEE International Joint Conference on Neural Networks, 1991, 3:2162–67.

[3] Acken K.P., Irwin M.J., Owens R.M.: A Parallel ASIC Architecture for Efficient Fractal Image Coding. The Journal of VLSI Signal Processing, 1998, 19(2):97–113(17).

[4] Howson I.: A Cost/Performance Study of Modern FPGAs in Cryptanalysis. Thesis, 2003, University of Sydeney.

[5] Watson J.D. and Crick F.H.C.: A Structure for Deoxyribose Nucleic Acid. *Nature* 171, (4356):737-738 , 1953.

[6] UniProtKB/Swiss-Prot, retrieved 5th April, 2012 from http://web.expasy.org/docs/swiss-prot_guideline.html

[7] GenBank, retrieved 5th April, 2012 from http://www.ncbi.nlm.nih.gov/pmc/articles/PMC540017/

[8] Durbin R., Eddy S.R., Krogh A., Mitchison G.: Biological Sequnce Analysis: Probabilistic Models for Proteins and Nucleic Acids. Cambridge University Press, Cambridge University UK, 1998.

[9] Salemi M. and Vandamme A.M.: The Phylogenetic Handbook: A Practical Approach to DNA and Protein Phylogeny. Cambridge University Press, 2003.

[10] Lehmann M. and Wyss M.: Engineering proteins for thermostability: the use of sequence alignments versus rational design and directed evolution. Curr Opin Biotechnol, 2001 Aug, 12(4):371-5.

[11] Allen M.P. and Tildesley D.J. "Computer Simulation of Liquids", Oxford University Press, 1987.

[12] Smith T.F., Waterman M.S.: Identification of common molecular subsequences. J. Molecular Biology, 1981, 147:195–197.

[13] Altschul S.F.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.Nucleic Acids Res 1997, 25:3389-3402.

[14] Saitou M. and Nei N.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol, 1987, 4:406-425.

[15] Altschul S.F., Gish W., Miller W., Myers E.W., Lipman D.J.. Basic local alignment search tool. J. Molecular Biology 1990, 215(3):403–410.

[16] Myes E.W. and Miller W.: Optimal alignments in linear space. Comput Appl Biosci, 1988, 4:11-17.

[17] Thompson J.D., Higgins D.G. and Gibson T.J.: CLUSTALW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res 1994. 22:4673–4680.

[18] Felsentein J.: Evolutionary trees from DNA sequnces: a maximum likelihood approach, J.Mol.Evol, 1981, 17:368-376.

[19] MrBayes software, retrieved 10th Augest, 2011 from http://mrbayes.sourceforge.net/download.php.

Chapter 2
An Introduction to GPU Computing

2.1 Introduction

Von-Neumann processors could be either general-purpose processors (GPPs), e.g. processors found on PCs, or special–purpose processors, e.g. digital signal processors (DSPs) or graphics processing units (GPUs). GPPs normally have a large amount of transistors used for logic control and cache sets, which have complex hardware architectures in order to perform complex operations such as speculative execution and branch prediction. GPPs are mainly designed for general-purpose applications, which range from simple data processing to complex mathematical calculations. Special-purpose processors aim to handle specific applications like digital signal processing and computer graphics processing. They have relatively less logic control units, but more arithmetic logic units (ALUs), which give them better performance in specific application domains. Although special-purpose processors are not as flexible as GPPs, they are more attractive for many applications. Since the architecture of special-purpose processors are specially designed to accelerate specific applications, e.g. graphics computing, significant performance advantages can be achieved compared to processors with general-purpose architectures. Moreover, special-purpose processors provide low energy consumption and lower cost for the performance they provide.

In this chapter, the essential background on parallel computing is first presented. Afterwards, fundamentals and characteristics of the widely used Graphics Processing Units (GPUs) are presented. The Compute Unified Device Architecture (CUDA) compatible GPU for general-purpose applications and its programming paradigm are then outlined. An overview of the memory model

on CUDA-compatible GPUs with compute capability 1.0 (the technology used throughout this thesis) follows before conclusions are laid out.

2.2 Essential Background on Parallel Computing

Computing models for arbitrary applications on a computer can be classified into sequential computing and parallel computing as a result of different execution strategies. Essentially, algorithms for applications are constructed and implemented as a serial stream of instructions in sequential computing systems. These instructions are implemented on a Central Processing Unit (CPU), and only one instruction can be executed at a time. After it is finished, other instructions are then implemented one by one. Pure parallel computing however utilizes multiple processors (e.g. in a multi-core processor) simultaneously to process a problem. Large tasks are often divided into small ones and assigned to multiple parallel processors. When these small tasks are independent of each other, they can be processed concurrently. Since computation requirements of modern applications are increasing year on year e.g. internet search engines, financial computing, data mining and scientific simulations, CPU manufacturers have traditionally improved the performance of CPUs by increasing CPU clock frequency and cache size. The reason for increasing frequency is based on the fact that the execution time of a program is equal to the number of instructions executed multiplied by the average execution time per instruction so that an increase in frequency can decrease the overall runtime. However, purely increasing the CPU frequency and cache size results an increase of the amount of dynamic power consumed by processors. Indeed, dynamic power consumption by a chip is given in Equation 2.1.

$$P = C \times F \times V^2; \quad (2.1)$$

where P is the dynamic power consumed, C is capacitance switched every clock cycle, V is the supply voltage and F is the clock frequency. In order to keep dynamic power manageable, CPU manufacturers relied on lowering the supply voltage V. However, as transistor feature sizes shrunk, supply voltage could not continue to be lowered as this implies lowering threshold voltage, which in turn increases static power considerably. The voltage wall in turn led to a frequency wall. The cancellation of Intel's Tejas and Jayhawk processors is generally cited as the end of frequency scaling as the dominant performance improvement strategy for computers [1]. CPU manufacturers had to find another way to keep doubling CPU performance every 18 months or so, the famous Moore's law [2].

Increasing the number of processors is another way to keep Moore's law going. Theoretically, doubling the number of processing cores should halve the runtime, but very few algorithms achieve optimal speed-up factors as most algorithms cannot be fully parallelized. Amdahl [3] presented a formula (see Equation 2.2) that is used to predict the potential speed-up factor of an algorithm on a parallel computing platform. It shows that a small proportion of program that cannot be parallelized will limit the overall speed-up of the program:

$$S = {}^{1}/_{A} \; ; \quad (2.2)$$

where S is the maximum speed-up factor, A is the faction of runtime a program spends on non-parallelizable (or serial) parts. For instance, the maximum speed-up achievable in an application with 10% serial (or sequential) code is 10x, no matter how many processors are added to implement the application. Amdahl's law assumes that the maximum speed-up factor can be achieved depends on the portion of non-parallelizable part, instead of the number of processors.

In addition to increasing the number of processing cores and the clock frequency of processors, there are other strategies to accelerate the execution time of applications which include word level and instruction level parallelism. The word level parallelism was developed to increase the computer word size, which reduces the number of instructions needed when the processors process variables whose size are greater than the native computer word. For instance, when an 8-bit processor is used to add two 16-bit integers, the processor has to first add the 8 lower-order bits of each integer using the native 8-bit addition instruction and then add the 8 higher-order bits using add-with-carry instruction with the carry-in bit from 8 lower-order bit additions. Hence, 8-bit processors need 2 instructions to complete 16-bit addition, while 16-bit processors are able to complete the operation in one instruction. Since the advent of Very Large Scale Integration (VLSI) computer fabrication technology to the advent of x86-64 architectures, word sizes of GPPs have experienced four stages: 4-bit, 8-bit, 16-bit, 32-bit and 64-bit respectively.

Moreover, improvements have been made to the execution model of instructions in computer programs. Traditionally, executing a steam of instructions by processors is done serially, i.e. the next operation would be performed only when the previous operation is completed. Modern processors have multiple stages of instruction pipelines, and each stage corresponds to a different operation. Processors with *N*-stage pipelines can process up to *N* different instructions at the same time as illustrated in Figure 2.1. The canonical example of a pipeline processor is a Reduced Instruction Set Computing (RISC) processor, where operations on instructions can be classified into five stages: instruction fetch, decode, execute, memory access and result write back. Using instruction level parallelism, the instruction throughput can be increased considerably (up to 5x in the canonical five-stage pipeline as illustrated in Figure 2.1(b)). However, in practice, cycles needed for filling and flushing the pipeline as well as instruction dependencies reduce this speed-up [4].

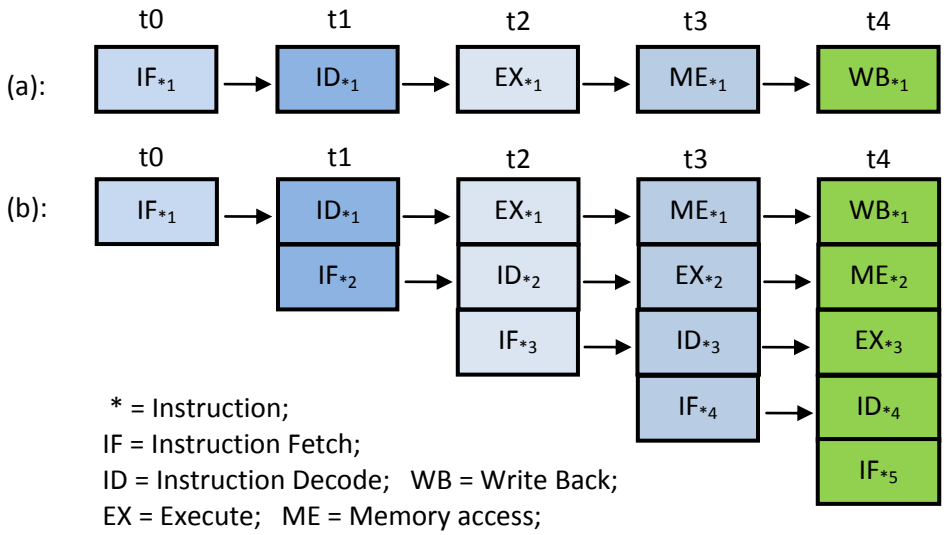


Figure 2.1: A traditional pipeline (a) and a canonical five-stage pipeline (b)

Flynn presented one of the earliest taxonomies of parallel computing systems, namely Flynn's taxonomy as shown in Table 2.1. These four classes are grouped according to whether the corresponding computer systems operate on single or multiple instructions, and whether or not these instructions operate on single or multiple data. Single instruction and single data (SISD) systems

Table 2.1: Flynn's taxonomy

	Single Instruction	Multiple Instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

are equivalent to the execution model of sequential programs. Single instruction and multiple data (SIMD) involves the execution of the same operation on multiple data in parallel, which is the execution strategy used in CUDA-compatible Graphics Processing Units (GPUs) as will be explained later in this chapter. Multiple instruction single data (MISD) involves performing

parallel instructions on the same data, and are relatively less used in practice. Multiple instruction multiple data (MIMD) involves multiple parallel instructions on multiple data and is very widely used in parallel computing such as in cluster computers, or even CUDA-compatible GPUs as will be explained later in this chapter.

2.3 Essential Background on Graphics Processing Units

The early 1980s is generally recognised as the period where modern computer graphics emerged. In 1981, the first video card was created by International Business Machine (IBM) Corporation. These cards were very basic as they only supported non-colour text. However, the emergence of such cards made computer solutions and personal computers (PCs) in particular more standard and popular. With the development of display technology, the capabilities of higher resolution, greater colour depth and the ability of controlling individual pixels on video cards made the task of CPUs more complicated. The idea of having an onboard processor dedicated for video processing was put forward to decrease the workload on the main system CPU. In 1984, IBM created the first processor-based video card for PCs. All video related work was processed by the Professional Graphic Adapter (PGA) and its own onboard processor, thus freeing the main system CPU from having to perform graphics computations. Silicon Graphics Inc. (SGI) then created the industry standards as the basis for today's computer graphics and the widely used platform-independent graphics API – OpenGL [5]. In addition to creating industry standards, the idea of graphics pipelines pioneered by SGI for the design of graphics hardware has been a major impetus for GPU technology. Nowadays' major GPU manufactures, including NVIDIA [6], ATI [7] and Matrox [8], are using the graphics pipeline approach to design their own graphics hardware. The most recent impetus behind the explosion of GPU technology has brought the rapid development of 3D game industry. Many classic 3D games, such as Quake series, Doom and flight simulators, have pushed high performance

GPUs into PCs at a phenomenal rate. Since GPUs are becoming more powerful, they are now being proposed to process high-end, supercomputer level applications, such as geophysical visualization applications [9], cloud simulation [10], financial computing [11], fluid dynamics [12] and *N*-body simulation [13].

Traditional general-purpose CPUs are single-threaded whereby multiple processes run through the same single-threaded pipeline with a single memory interface. GPUs, on the other hand, have a completely different architecture, which is based on steam processing. A GPU can utilize tens or hundreds of steam processors to complete graphics rendering. The rendering pipeline is composed of vertex shaders, geometry shaders and pixel shaders respectively. Shaders are used to process a large set of elements at a time and transfer these elements into other shaders for other functions. The process of rendering graphics in GPU can be illustrated in Figure 2.2. The GPU first reads the vertex data which is used to describe the 3D appearance of the mountains from GPU memory. After that, the GPU uses these vertex data to determine the shape and location of the 3D graphics for building the 3D shape. The shape is then divided into multiple triangles, which are processed on Transform and Lighting operations by corresponding elements in GPU. The fundamentals of GPUs will be presented in the next sub-section.

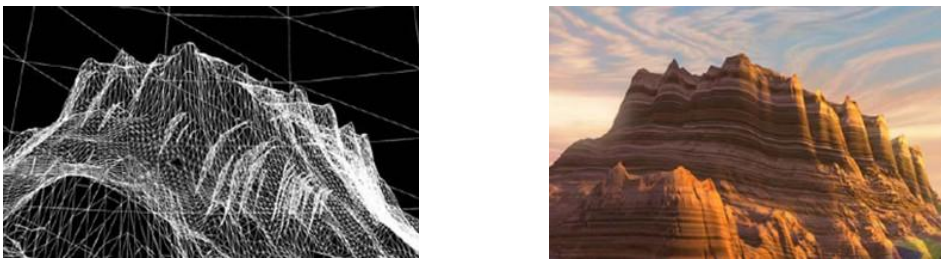


Figure 2.2: An Example of Mountains Rendering

2.3.1 Fundamentals of Graphics Processing Units

Graphics Processing Units (GPUs) are essentially a dedicated hardware device designed to rapidly manipulate the acceleration of building images in frame buffers for display. GPUs are very efficient at processing computer graphics because of their highly parallel architecture. In 1999, NVIDIA Corp. developed the pioneering GeForce 256 GPU which is a single-chip processor that is capable of processing a minimum of 10 million polygons per second. In general, GPUs are responsible for translating objects into 2D images formed by pixels. A 3D scene on the screen composed by different objects in certain locations is actually transformed by many 2D images. Each object on the screen, whether it is a wall, floor or others, is a combination of a certain number of triangles. Therefore, arbitrary complex objects in 3D space can be split into multiple triangles, and the latter are composed of vertices[1] and textures[2]. There are many different representations of the concept of a 3D pipeline, some of which are very simple and some are very complex. The whole process can be summarized into 2 stages. The first one is the geometry processing stage, which transforms 3D object coordinates into 2D coordinates. The second one is Rasterization stage which fills colours in those 2D images. The Graphics Pipeline of GPUs is illustrated in Figure 2.3. In the graphics pipeline, the geometry stage is also referred to as the Transform and Lighting (T & L) stage. For the purpose of transferring the 3D scene to 2D image, all the objects in the scene need to be transformed to various spaces, and each has its own coordinate system. The lighting operation is used to add light on the surface of objects, which is achieved by calculating the relationships between the position of view point and the position of the light source. After that, the Rasterization stage converts data into a number of fragments, which may become pixels of the final image later. Figure 2.4 illustrates the Rasterization process of a triangle. Note that if the fragment is intersected with other primitives,

[1] A vertex is a point with spatial, colour and texture coordinates
[2] A texture is an image which is used to map onto the surface of objects

additional calculations are needed to interpolate the attributes between the primitives. In addition to filling colour on objects, texture creates the illusion of an object consisting of a certain material, while vertices of an object specify how a texture is mapped onto any given surface of objects. Figure 2.5 illustrates the process of mapping a wall texture onto the surface of a triangle object.

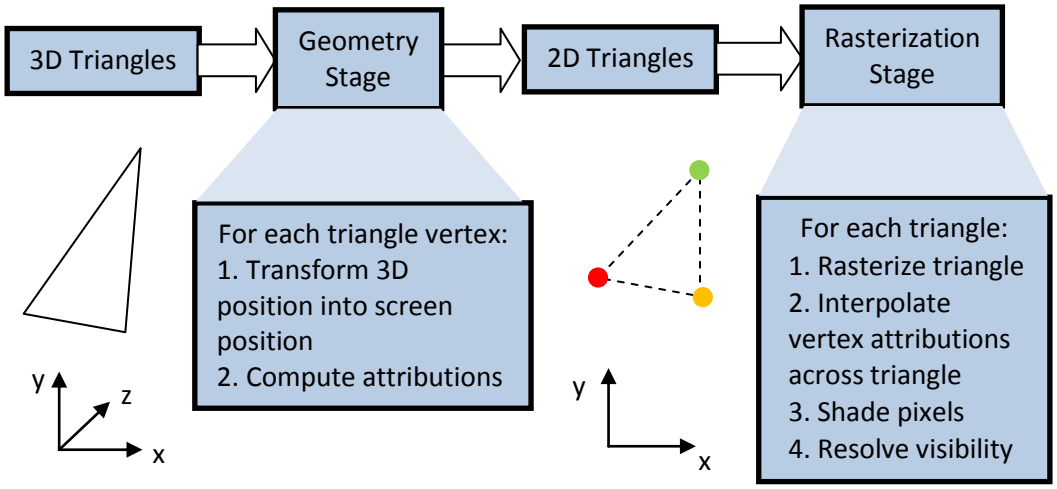


Figure 2.3: A 3D Graphics Pipeline

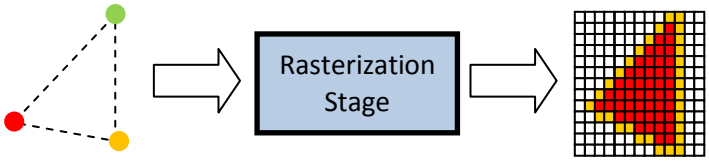


Figure 2.4: Rasterizing a triangle

Figure 2.5: Mapping a texture into a triangle

2.3.2 Characteristics of Graphics Processing Units

In a modern graphics pipeline, GPUs perform most graphics related tasks such as T&L operations, triangle setup and rendering, hence freeing the main system CPU. However, GPUs were first designed for the purpose of rendering objects as it was relatively simple to render objects with correct colours. For drawing a line, the rendering engine fills the colour from the starting to the ending point of the line. For drawing a triangle, the previous process is repeated to fill all pixels that make up the triangle. When the task of rendering a triangle with 10000 rows in software, the CPU has to compute and transfer 1000 different sets of starting and ending points of the triangle to the graphics card for rendering. Since this task requires massive computations when there are millions of triangles waiting for setup, the CPU becomes the bottleneck in the graphics pipeline. Thus, the triangle setup stage was transferred to GPUs. However, allowing GPUs to process the triangle setup stage raises a new problem of how to speed up the graphics hardware to catch up with the number of triangles that the CPU outputs. This in turn led to the concept of graphic pipeline on GPUs. A graphics pipeline processes many pixels at the same time by one of its components and quickly passes these pixels to the next component. In addition to the use of graphics pipeline and since the work of rendering 3D graphics has extremely repetitive operations, GPUs have been adorned with many pipelines to allow for large data parallelism. However, this transferred the bottleneck in the 3D graphics hardware to CPUs as these may not provide sufficient data to keep graphics hardware busy. Given that the transistor count doubles every 6 months for a GPU and every 18 months for a CPU [14], GPUs were augmented to take on the step of the 3D graphics pipeline, namely transform and lighting. Figure 2.6 illustrates how GPUs have taken on the tasks of 3D graphics pipeline over time [15]. Note that it was in 1999 that NVIDIA implemented the step of Transform and Lighting in 3D graphics pipeline in hardware with their GeForce 256 GPU.

Application tasks	CPU	CPU	CPU	CPU
Scene Level calculations	CPU	CPU	CPU	CPU
Transform	CPU	CPU	CPU	GPU
Lighting	CPU	CPU	CPU	GPU
Triangle Setup and Clipping	CPU	Graphics Processor	Graphics Processor	GPU
Rendering	Graphics Processor	Graphics Processor	Graphics Processor	GPU
	1996	1997	1998	1999

Figure 2.6: NVIDIA 3D graphics Pipeline Timeline [15]

This progress in graphics rendering made GPUs do all graphics related tasks, instead of having to rely on the main system CPU for some tasks, e.g., Transform and Lighting operations.

The initial graphics pipeline was a fixed function pipeline. Once the graphics data is transferred into GPUs, programmers could not modify data as they desire. For instance, when a fixed function pipeline draws a complex 3D scene, the vertices of all objects in the scene need to be passed into the vertex shader at the same time, which means their shapes are fixed and no further input on how hardware creates the final image was possible. Hence, the problem behind the fixed function pipeline is the inflexibility of graphics operations which hampers creativity. Developers were indeed limited to the specific set of features that are defined by the supported software, i.e. DirectX and OpenGL. In 2001, NVIDIA GeForce 3 was introduced. This new generation allowed limited amount of programmability in the vertex pipeline. Indeed, instead of just sending positions and colours to GPUs, developers could control vertex and pixel shaders to interpret the data before creating the final image. In 2002, GeForce FX was introduced and considered to be the first generation of fully-programmable graphics cards. In it, vertex shader programs operate at the vertex level and replace the transform and lighting (T&L) stage of the 3D graphics pipeline while pixel shader programs operate at the pixel level and replace the Rasterization and interpolation stages of the pipeline. The

difference between the graphics pipeline of GeForce 256 and GeForce FX is illustrated in Figure 2.7.

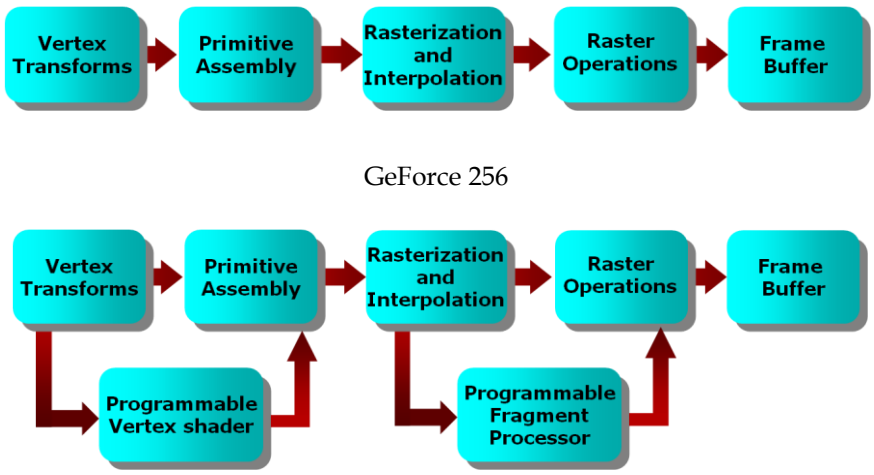


Figure 2.7: Graphics Pipeline of the GeForce 256 and GeForce FX GPUs

As discussed above, the stages of the GPU 3D graphics pipeline have evolved from software implementations on the main system CPU to a fixed function graphics pipeline on a graphics co-processor, and then to a programmable pipeline on a graphics co-processor. Since modern graphics pipelines are becoming more and more programmable and flexible, GPUs not only are optimized for highly parallel graphics processing, but also can be thought of as general-purpose stream processors, instead of processing graphics related tasks only. However, programmable GPUs face a problem when they process more vertices and fewer graphics operations, or more graphics operations and fewer vertexes. Such scenarios lead to data load imbalance as the number of vertex shaders and pixel shaders are fixed. For instance, suppose there are totally 5 vertex shaders and 5 pixel shaders on a GPU. Suppose also there is a complex scenario including thousands of vertices but only 1 colour used for rendering. Here, it is easy to see that the 5 vertex shaders are in relatively busy states. The 4 pixel shaders on the other hand are often in idle state, which

decreases the performance of the overall GPU pipeline. NVIDIA put forward a new generation of GPU architectures – called Compute Unified Device Architecture (CUDA) to overcome this problem, as will be explained in the next section.

2.4 Compute Unified Device Architecture

In 2006, the world's first fully unified shader for GPUs, namely GeForce 8800 GTX, was introduced. This signalled the 8th generation of NVIDIA's GeForce line of GPUs. The new architecture is called Compute Unified Device Architecture (CUDA). The first CUDA-compatible GPU consisted of 128 steam processors (SPs), and each 8 of them are grouped into one Stream Multiprocessor (SM). Unlike the vector processing approach taken with older types of shader units, each stream processor is capable of being dynamically allocated to a vertex, pixel, geometry or physic operations for the utmost efficiency in GPU resource allocation and maximum flexibility in load balancing shader programs [16]. Figure 2.8 illustrates the difference between the classic GPU architecture and unified shader architecture.

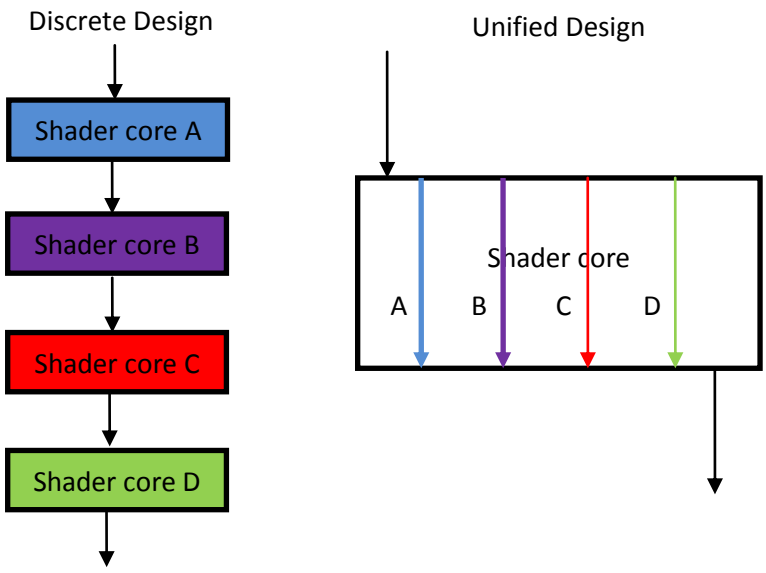


Figure 2.8: A classic GPU vs. A unified shader GPU [16]

As shown in the figure 2.8, the classic graphics pipeline uses discrete shader types represented in different colours, while the illustration on the right depicts a unified shader core with one or more standardized, unified shader processors. Data flow in unified designed shader core is dispatched into the relevant shader processor until all shader operations are performed.

2.4.1 The Classic Graphics Pipeline vs. CUDA

For a classic GPU, data is sequentially processed by different types of shader. Once the process of one data type is not finished, the entire data flow cannot be passed into the next shader type. In addition, for vertex shader-intensive or pixel shader-intensive scenarios, the fixed number of shader units for each type becomes the bottleneck on performance. This can be illustrated in Figure 2.9. For scene *A* with heavy geometry, while vertex shaders are all fulfilled, but fewer pixel shaders have workload. Scene *B* with heavy pixel however requires fewer shaders on vertex processing, but more shaders on pixel processing. If measuring performance by the maximum number of the vertex shaders, the maximum performance can be attained is 4 and 8 respectively. Therefore, both situations are not optimal as graphics hardware is often idle.

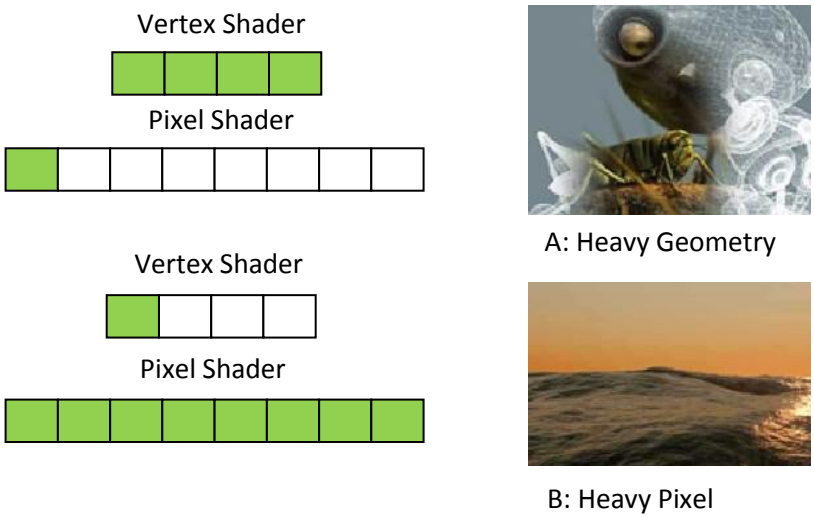


Figure 2.9: Fixed shader performance characteristics [16]

The unified shader designed GPU however integrates multiple shader types into one core so that arbitrary data type can be processed in the same shader core. Therefore, all shader cores in unified shader designed GPUs can be kept busy at any moment as long as there is enough data to process, no matter whether the application is vertex shader-intensive or pixel shader-intensive. The benefit of this architecture can be illustrated in Figure 2.10. For scene *A*, 11 vertex shaders represented in green colour process vertex data and 1 pixel shader represented in red colour process pixel data. Similarly, for scene *B*, 11 pixel shaders represented in red colour process pixel data and 1 vertex shader represented in green colour process pixel data.

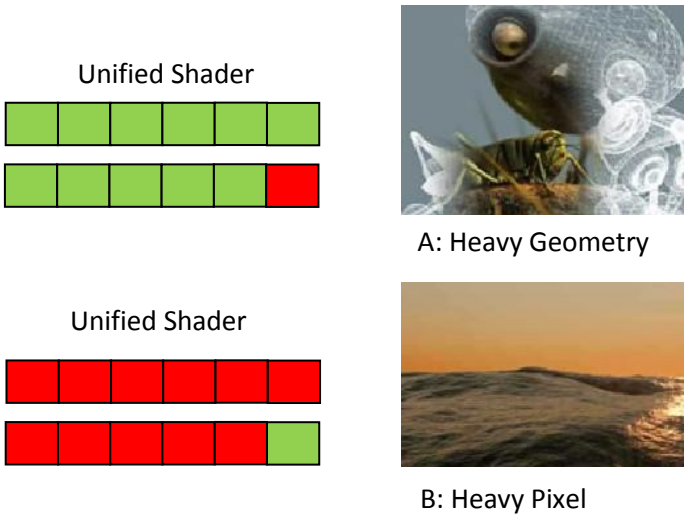


Figure 2.10: Unified shader performance characteristics [16]

In the next sub-section, the essentials of the CUDA programming Model, introducing the thread batching strategy will be presented firstly. After that, the memory model of CUDA-compatible GPUs is then presented.

2.4.2 CUDA Programming Model

As discussed above, data-parallel or compute-intensive parts of applications running on the main system CPU can be off-loaded onto the GPU device. The development of CUDA-compatible GPUs not only aims to make the processing of graphics related tasks more efficient but also general purpose compute-intensive applications. When programming in CUDA, the GPU can be thought of as a highly multi-threaded coprocessor. Applications which operate on different data independently can readily benefit from the GPU parallelism by executing many different threads on different data. GPU functions are packaged in sets of instructions called 'kernels'. A kernel is downloaded to the device in advance before running it using multiple threads. The batch of threads that implements a kernel is grouped as a grid of thread blocks. A thread block is a batch of threads which can cooperate together by efficiently sharing data through fast shared memory access. On the other hand, different threads in the batch can be in charge of different tasks to complete the entire kernel application. These two task allocation strategies are called intra-task and inter-task parallelization strategies, respectively. Each thread in a thread block is identified by its thread ID, which is calculated by its index *(x, y, z)* in a three-dimensional thread block. Parameters x, y and z cannot exceed the corresponding dimension size of blocks. More precisely, for a two-dimensional block of size (D_x, D_y), the thread ID of a thread with index (x, y) is equal to ($x + y\ D_x$). For a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread with index (x, y, z) is equal to ($x + y\ D_x + z\ D_xD_y$). Since the maximum number of threads that a block can contain is limited, there are normally many thread blocks allocated to process an application. Given an application, once a thread block completes its task, the remaining tasks are distributed to other blocks in the same grid. Similarly, each block in a block grid is identified by its block ID calculated by its index *(x, y, z)* in a three-dimensional block grid. The relationships among thread, block and grid are illustrated in Figure 2.11 for two and one dimensional grids and two dimensional blocks. Each kernel

corresponds to a block grid. The size of blocks and grids needs to be defined in variables *dimBLOCK* and *dimGrid*, respectively, before launching kernels.

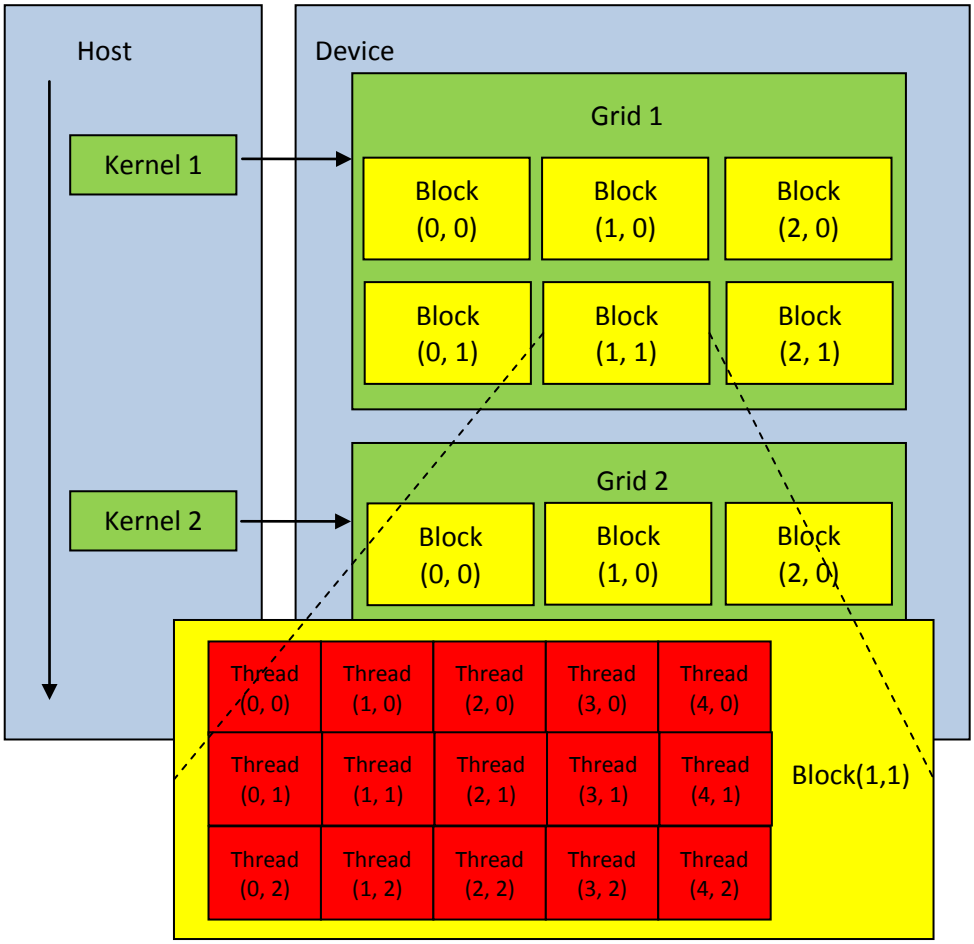


Figure 2.11: Each kernel is executed as a batch of threads organized as a grid of thread blocks

The execution of a grid of thread blocks on GPU is performed by scheduling blocks on the SMs. Each SM is able to process one batch of blocks at the same time, and processing blocks are referred to as active blocks. After completing one batch, the SM then processes another one until the completion of all batches of blocks. The number of blocks in each batch depends on the number of registers required per thread and the size of shared memory required per block for a given kernel. If there are not enough registers or shared memory per multiprocessor to process at least one block, the kernel will fail to launch.

The threads in each block are split into SIMD groups called warps, which are sets of consecutive threads, e.g. the first warp contains threads with IDs 0 to *N-1*, whereas the second warp contains threads with IDs *N* to *2N-1* etc. The number of threads contained in each warp is called the warp size (32 in GeForce 8800 GTX GPUs). Threads in the same warp are executed by the SM in a SIMD fashion. All warps from all active blocks are executed in a MIMD fashion: multiple instructions are executed by multiple warps and threads in the same warps execute the same instructions but for different data. The schedule among active warps is time-sliced. The maximum number of active warps that can be scheduled is dictated by the SM's computational resources, such as shared memory and registers. The issue order of warps in the same block is not transparent to users, but their execution can be synchronized. Hence, it is safe to perform a read or write operation from/to global memory in the same block after thread synchronization. However, there is no synchronization mechanism between blocks, so threads from different blocks cannot safely communicate with each other through global memory. The safest approach is to make different warps perform computations for independent tasks. Within warps, however, and because of their SIMD execution fashion, thus there is no need to perform thread synchronization. Normally, the total number of threads within a thread block should be equally distributed into several complete warps, to avoid thread load imbalance.

2.4.3 Memory Model

The device memory space of various scopes is illustrated in Figure 2.12. A thread that executes on the GPU has only access to the following type of memory space on device. Understanding the usage of each kind of memory space is crucial to maximise the application on GPUs. More details about the memory space on GPU device are given below:

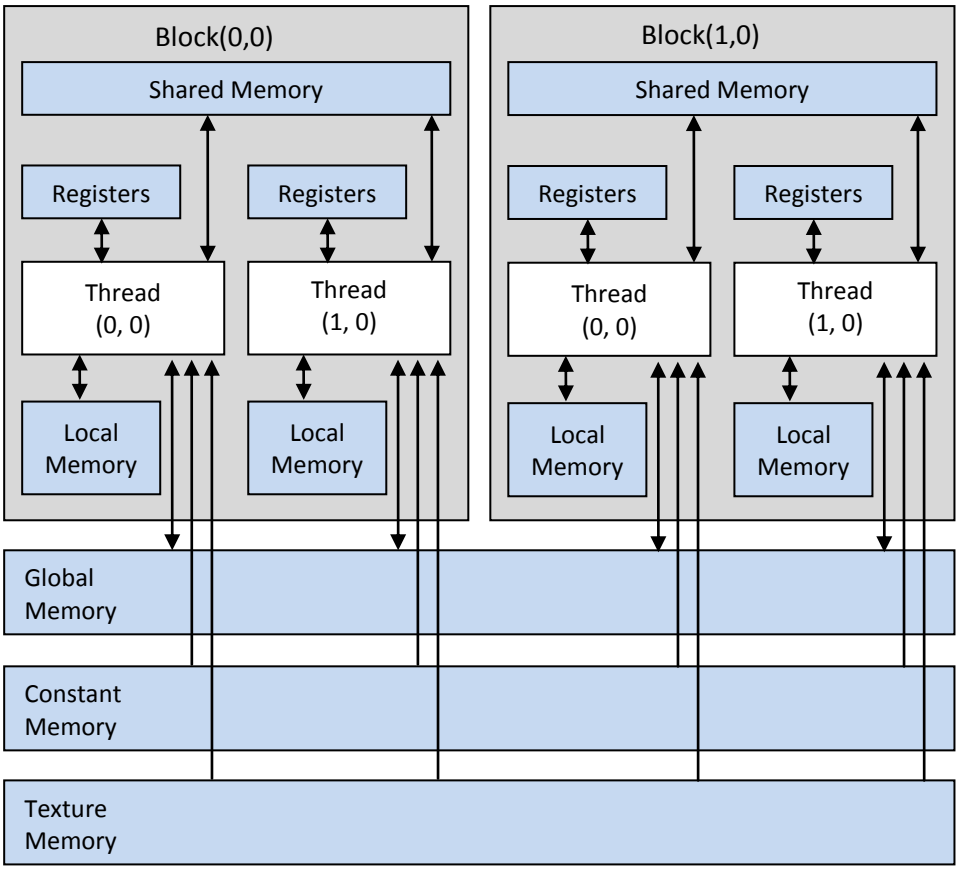


Figure 2.12: The GPU device memory and on-chip memory spaces

- Registers: this is on-chip memory meaning it resides on the stream multiprocessors, which makes it very fast (1 memory clock cycle). It is however limited in number (e.g. 8192 registers in each SM on GeForce 8800 GTX). Registers can be read and written to by threads.
- Local memory: this is off-chip memory, meaning it resides outside the stream multiprocessors on GPU device memory. It is not cached, and is local in the scope of each thread. Local memory can be read and written to by threads.

- Shared memory: this is on-chip memory which means its access time as fast as registers if there is no bank conflict. Shared memory can be read and written to by threads.
- Global memory: this is off-chip memory which is not cached. Threads in the same half-warp can access global memory if memory access can be coalesced into a single contiguous and aligned memory. Global memory can be read and written to by all grid threads.
- Constant memory: this is off-chip memory which is cached. It is optimized for the case when many threads read the same memory location. If threads read from multiple locations, access is serialized. Constant memory can be only read by grid threads.
- Texture memory: this is off-chip memory which is cached. It is optimized to accelerate frequently executed operations. Texture memory can be only read by grid threads.

One of the most important performance considerations in global memory access is "Coalesced Global Memory Access", as global memory access for reading 32-bit, 64-bit or 128-bit words can be performed in single read or write operation by threads within a half-warp, if access follows the right access. Indeed, performance could decrease to one sixteenth if threads access misaligned memory positions. More precisely, suppose half-warp threads read values from an array composed by 16 different sequences of 16 residues each (i.e. 16 rows and 16 columns). Suppose also that the 1st residue *A[x][1]* to the 16th residue *A[x][16]* needs to be fetched from these sequences (*x* stands for the 16 sequences). The total number of accesses by half-warp threads is 16 if we store the 16 sequences in global memory as shown in Figure 2.13. In this way, each request from half-warp threads to global memory can be reduced to a single load instruction as the addresses involved are continuous. The type in Figure 2.13 must be such that its size is equal to 4, 8 or 16 bytes corresponding

to 32-bit, 64-bit or 128-bit, respectively. The addresses presented to the memory controller are shown in Fig 2.13(a).

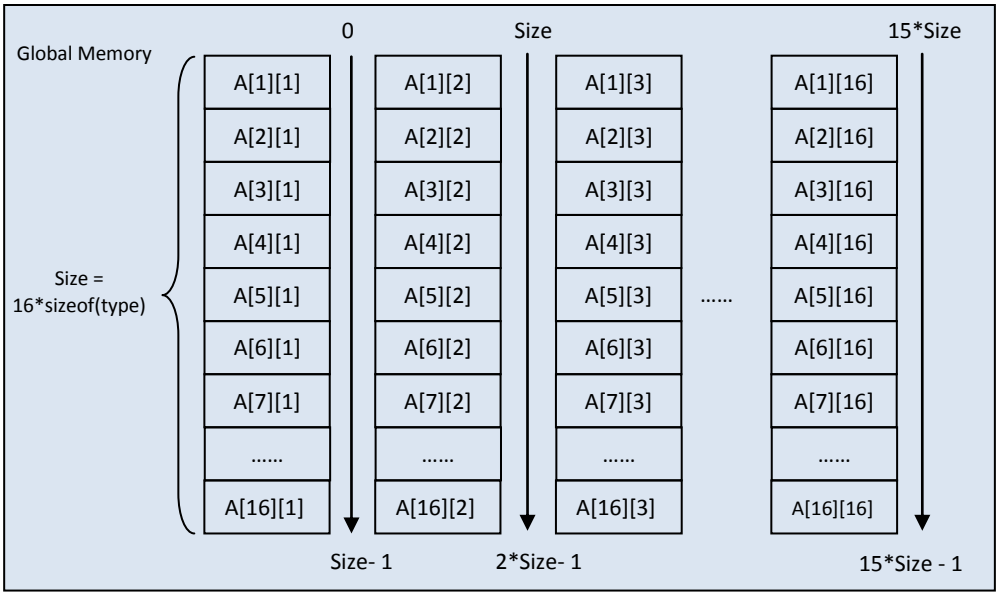


Figure 2.13: Aligned memory for coalesced memory access

The performance could decrease to one sixteenth if threads access the memory pattern as illustrated in Figure 2.14. In this case, each request from half-warp

threads to global memory needs to be compiled into 16 load instructions. For instance, the address of the 1st residue in the 16 sequences is discrete with an offset value of Size, thus the memory controller has to split the request into 16 requests as shown in Fig 2.14(a):

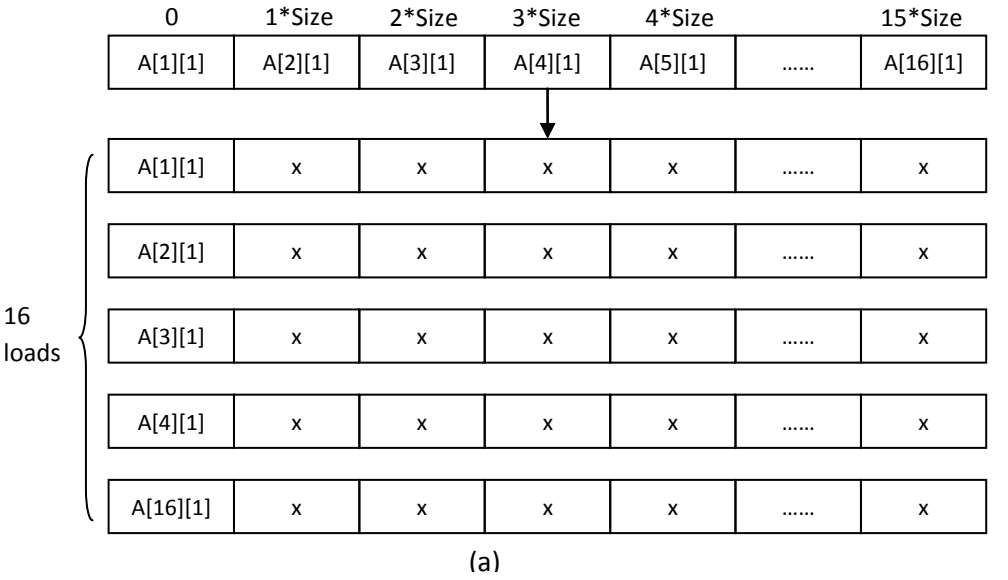


Figure 2.14: Misaligned memory that cannot achieve coalesced memory access

Each request to fetch 16 residues is translated to 16 individual fetches performed serially. Therefore, the overall load requests are 256, instead of 16 as discussed in the coalesced memory pattern, which is a heavy performance hit.

Moreover, usually, there are many arrays allocated in global memory. Thus the base addresses for arrays also need to be considered. If the first array allocated in global memory is misaligned, the requests to the following allocated arrays still cannot be coalesced. In general, in each half-warp, thread *N* within the half-warp should access address:

$$HalfWarpBaseAddress + N$$

where *HalfWarpBaseAddress* should be aligned to 16*sizeof(type) bytes, i.e be a multiple of 16*sizeof(type). Since the type could be 4-byte, 8-byte or 16 byte, the minimum array size *N* should be at least 64 bytes to satisfy the memory alignment constraint. If half-warp threads fulfil the requirements above, the per-thread memory accesses still will be coalesced even if some threads of the half-warp do not actually access memory, i.e. divergence.

In addition to the coalesced access of global memory, the use of shared memory is another important performance consideration. Since it is on-chip, access to shared memory space is much faster than global memory space. In fact, accessing the shared memory is as fast as accessing registers as long as there are no bank conflicts between threads of the same warp. Shared memory is separated into equally-sized memory banks to achieve high memory bandwidth. Any memory read or write request to *n* addresses which fall in *n* distinct memory banks can be performed simultaneously. However, if several addresses of a memory request fall in the same memory bank, a bank conflict occurs, and access has to be serialized. For devices of compute capability 1.x, the number of shared memory banks is 16. In GeForce 8800 GTX, there are 16k bytes of shared memory for each SM. This is divided into 16 equally-sized banks, and each bank has 32-bit word length and can store 256 integers or floating points. Figure 2.15 illustrates the architecture of the shared memory in Geforce 8800 GTX. To allow for parallel access, an array of 48 integers, for instance, is assigned to 16 banks separately from bank 0 to bank 15 iteratively.

The gray ones in the figure denote the first 16 integers, the green ones denote the mid 16 integers, and the blue ones denote the last 16 integers.

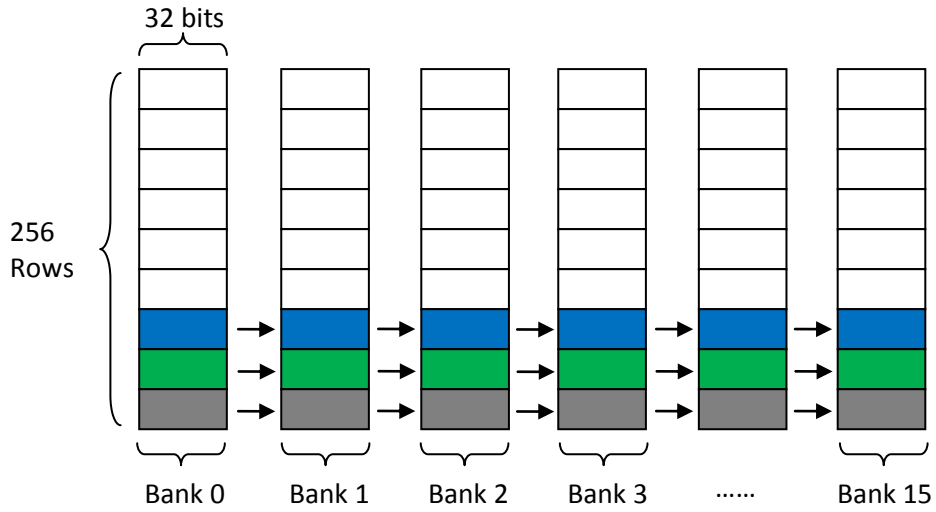


Figure 2.15: The Architecture of shared memory in GeForce 8800 GTX

A request of a warp threads to shared memory is split into 2, with each request for a half-warp threads. A bank conflict occurs when part of the threads in half-warp fall in the same bank, which reduces the efficiency of memory access. The most serious case is when 15 requests of half-warp threads fall in the same bank (Recall that half-warp size is 16). Half-warp threads access to the same address however results in a memory broadcast with access speed as fast as accessing registers. More precisely, suppose having an integer array *data[128]* in shared memory. The successive access approach below does not lead to bank conflicts and achieve the maximum performance:

int number = data[base + tid];

This approach sometimes does not suit other data types, such as char or structures. Given a char-type array, four successive characters are stored in a bank, hence access to shared memory for a half-warp fall in 4 banks. In addition to the constraint of data type, the discrete access approach as

illustrated in Fig 2.16 also leads to bank conflicts, as thread 0 and 4 falls in the same bank (bank 0), thread 1 and 5 falls in bank 4 and so do the other threads. Therefore, there are 4 threads in half-warp accessing the same banks, which leads to a 4-way bank conflict as shown in below:

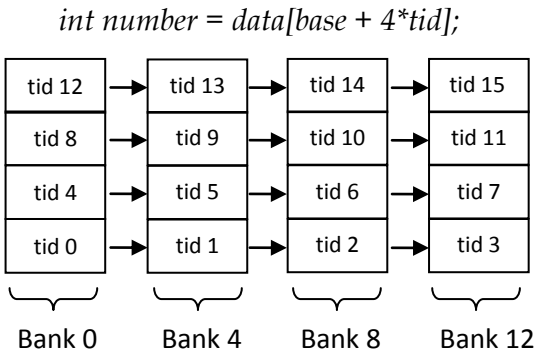


Figure 2.16: An example of bank conflicts for accessing shared memory

Having presented the CUDA programming model and corresponding memory model, the next sub-section gives a brief overview of the GPU used in this research project.

2.4.4 The GeForce 8800 GTX GPU

The GPU used throughout this research project is NVIDIA's Geforce 8800GTX, which has 16 Stream Multiprocessors (SMs), with each SM having 8 Stream Processors (SPs) used as Arithmetic Logic Units (ALUs). In each SM, there is 8KB on-chip constant memory, 8KB on-chip texture memory, 16KB on-chip shared memory and 8192 registers. The off-chip device memory offers 768 Mbytes memory space, which contains global memory, texture memory and constant memory. The corresponding architecture is illustrated in Figure 2.17.

In addition to above, the general specifications of GeForce 8800 GTX are presented in below:

- The maximum number of threads per block is 512
- The maximum sizes of the x-, y-, z-dimension of a thread block are 512, 512, and 64, respectively.

- The maximum size of each dimension (x, y, z) of a grid are 65535.
- The maximum number of active blocks per SM is 8.
- The maximum number of active warps per SM is 24.
- The warp size is 32 and half-warp is 16.

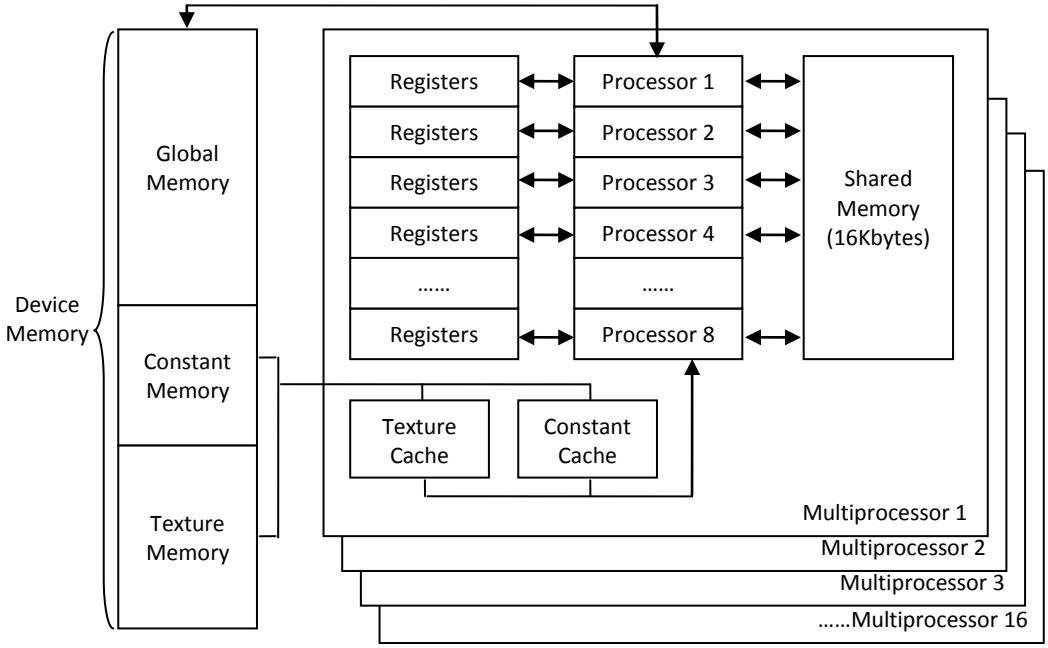


Figure 2.17: The Architecture of NVIDIA's GeForce 8800 GTX

2.5 Conclusions

In this chapter, the essential background on parallel computing was first presented. Through analyzing the problem of increasing CPU frequency by manufacturers to achieve the Moore's law, the GPU technology as an alternative hardware acceleration platform for specific applications was laid out. Moreover, the evolution of graphics pipelines on GPUs from a pure software implementation to powerful unified and programmable 3D graphics hardware was presented, thus GPUs can now be used as general-purpose processors. In addition to their powerful computing ability, GPUs are simpler and cheaper to develop than many other kinds of processors, as they benefit from the economies of scales of the gaming industry. Many implementations have shown that their computations can be easily accelerated to one order of magnitude by simply investing a few hundred dollars on a commercial graphics card to do the computation. Although GPUs currently cannot substitute CPUs as the main system processors, it still outperforms CPUs in solving massive, parallelizable problems.

2.6 References

[1] Flynn and Laurie J.: Intel Halts Development of 2 New Microprocessors. The New York Times, 2004.

[2] Moore's Law to roll on for another decade, http://news.cnet.com/2100-1001-984051.html. Retrieved 2012-2-10.

[3] Amdahl G.M.: The validity of the single processor approach to achieving large-scale computing capabilities. In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, pp.483–85.

[4] Mano, M. Morris (October 1992). Computer System Architecture (3rd ed. ed.). Prentice-Hall. ISBN 0131755633.

[5] Segal M., Akeley K. and Frasier C.: The OpenGL Graphics System: A Specification. Silicon Graphics, 1998.

[6] NVIDIA Corporation, NVIDIA homepage retrieved 1st Jan, 2012 from http://www.nvidia.com/page/home.

[7] ATI, ATI homepage retrieved Feb 2012 from http://www.ati.com/.

[8] Matrox, Matrox homepage retrieved Feb 2012 from http://www.matrox.com/.

[9] NVIDIA Corporation: 3D Graphics Demystified, retrieved Feb 2012 from http://developer.nvidia.com/content/technical-brief-3d-graphics-demystified.

[10] Wang k. and Shen Z.: Artificial Societies and GPU-Based Cloud Computing for Intelligent Transportation Management. Intelligent Systems, IEEE, 26(4):22-28, 2011.

[11] Lee M., Jeon J.H., Kim J. and Song J.H.: Scalable and Parallel Implementation of a Financial Application on a GPU: With Focus on Out-of-Core Case. IEEE 10th International Conference on Computer and Information Technology (CIT), 26(4):1323-1327, 2010.

[12] Zhao X.K., Li F.X., Zhan S.Y.: A New GPU-Based Neighbor Search Algorithm for Fluid Simulations. 2nd International Workshop on Database Technology and Applications (DBTA), pp.1-4, 27-28, 2010.

[13] Jetley P., Wesolowski L., Gioachin F., Kale L.V. and Quinn T.R.: Scaling Hierarchical N-body Simulations on GPU Clusters. 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp.1-11, 13-19, 2010.

[14] Moller J.: Real‐Time Rendering. A. K. Peters, 2002.

[15] NVIDIA Corporation: Technical Brief: Transform and Lighting, retrieved Feb 2012 from http://www.nvidia.com/object/Technical_Brief_TandL.html.

[16] NVIDIA Corporation: Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview, retrieved Feb 2012 from http://www.nvidia.com/page/8800_tech_briefs.html.

Chapter 3
A Parameterisable Smith-Waterman Algorithm Implementation on CUDA-compatible GPUs

3.1 Introduction

Biological sequence alignment is a widely used and fundamental operation in the field of bioinformatics and computational biology. It aims to find out whether two or more biological sequences are related or not. However, standard biological sequence alignment tasks are also computationally expensive as their computing and memory requirements grow quadratically or even faster with the size of biological sequence databases. Given that the latter are growing at exponential rates year after year, the need for hardware acceleration is getting stronger [1].

While special-purpose hardware, such as super-computers, can be used to alleviate the above problem, the resulting cost cannot be afforded by general users. For these reasons, heuristic approaches have been developed to run sequence alignment tasks at reasonable costs at the expense of some loss of accuracy. FASTA [2] and BLAST [3] are prime examples of such heuristic, and result in up to 50x speed-up compared to the widely used Smith-Waterman algorithm running on general purpose computers. Instead of using heuristic approaches, the Smith-Waterman algorithm harnesses exhaustive search approach which is guaranteed to find the optimal local alignment.

Graphics Processing Units (GPUs) have been recently proposed as high performance and relatively low cost acceleration platforms for biological sequence alignment [5]. A GPU-based implementation of the Smith-Waterman algorithm by Liu et al. [6] was one of the first reported GPU implementations of the Smith-Waterman algorithm. After that, Manavski et al. [7] and Munekawa et al. [8] reported two implementations of the Smith-Waterman

algorithm with two different thread parallelization strategies, targeting NVIDIA GPUs [9]. The former used a single thread to calculate a complete pair-wise alignment matrix, column by column serially, so that many threads can perform the computation of alignment matrices of different pairs in parallel. The latter implementation harnessed a batch of threads to compute a alignment matrix in parallel, exploiting the fact that the computation of the matrix elements on each anti diagonal are independent of each other, and hence can be done in parallel. Both implementations used the compute unified device architecture (CUDA) application programming interface (API) to program and target GPUs. The API have contributed greatly to the popularity of GPUs in general purpose computing, opening the way for a new field of computational study coined general-purpose computation GPU (GPGPU) which aims to harness the computational power and low cost of GPUs for a wide range of applications beyond graphics, including scientific computing [10], computational geometry [11], image processing [12] and bioinformatics.

Compared with CPU implementations of the Smith-Waterman algorithm, e.g. from Farrar [13] and SSEARCH [4], implementations from Manavski et al [7] and Munekawa et al [8] demonstrated considerable acceleration performance, which achieved up to 30x speedup. However, both implementations have a serious limitation on the length of query sequences that their GPU implementations can cope with. Indeed, Munekawa et al clearly stated that that query sequences should be shorter than 2048-residue long, because of the limitation on the maximum number of threads that could be defined in each block and the size of the built-in variable, i.e., 512 × 4. Manavski reported a similar limitation because of the limited size of the local memory. Such limitation renders these implementations useless in many real world applications where query sequences are far longer than 2048. In this chapter, a technique which overcomes the above limitation of previous implementations of the Smith-Waterman algorithm is presented. The main idea behind this is to separate the computation of the alignment matrix into multiple parts if the

number of threads and size of local memory are not sufficient, and allocate the available resources to each sub-matrix in turn.

The remainder of the chapter is organized as follows: Firstly, essential background information on biological sequence alignment and the Smith-Waterman algorithm are presented. Then, previous work in the area of GPU-based acceleration of biological sequence alignment is presented. Afterwards, our novel GPU-based implementation technique of the Smith-Waterman algorithm and pseudo code are presented. A comparative evaluation of our implementation then follows. After that, our own implementations of the Smith-Waterman algorithm using both intra-task and inter-task parallelization approaches for four specific database sets and evaluate the difference between these two approaches are presented. Finally, conclusions are laid out.

3.2 Background - Essentials of the Smith-Waterman Algorithm

Mutation, selection and random genetic drift are the normal evolution methods of biological sequences. Mutation particularly changes sequences in three main processes. Consider DNA sequence {A, T, T, C, G} as the original sequence, if one mutation occurs, it would be one of the following three alternatives:

- Substitution of residues: suppose the second residue A is substituted by G, then {A, T, T, C, G} would mutate to {A, G, T, C, G}.

- Insertion of residues: suppose there is a new residue C inserted between A and T, then {A, T, T, C, G} would mutate to {A, C, T, T, C, G}. The alignment of the original sequence to the new one could be expressed as {A, -, T, T, C, G}, where '-' denotes an insertion.

- Deletion of residues: suppose the existing residue G is deleted, then {A, T, T, C, G} would mutate to {A, T, T, C}. The alignment of the new sequences to the original one could be expressed as {A, T, T, C, -}.

Gap '-' appears in alignment when insertions or deletions occur. The appearance of gaps is for the purpose of making identical or similar residues align in successive columns so that the degree of two aligned sequences can be measured by the sum score of all the aligned characters. A substitution matrix, such as BLOSUM50, describes the evolution rate at which one character in a sequence evolved from other character states over time. It is used to assign scores for all substitutions in protein sequences, while gap penalties also contribute to the overall score of alignments. The commonly used types of gap penalties are linear gap penalty and affine gap penalty:

- Linear gap penalty: which has one constant penalty cost d, and it is the cost per unit length of gap. Suppose the length of gap is N, the resulting linear gap penalty is calculated by Equation 3.1.

$$Penalty = d \times N \quad (3.1)$$

- Affine gap penalty: which is an extension to linear gap penalty, and it has two parameters: a gap opening penalty o and a gap extension penalty e. Typically, o is larger than e, and the commonly used values are 10 and 2, respectively. Affine gap penalty is preferred to linear gap penalty as it is more biological accurate. Indeed, in practice, once a gap is open, it is usual to have several consecutive gaps. Afterwards, the overall cost for consecutive gaps with length N can be calculated by Equation 3.2.

$$Penalty = o + N \times e \quad (3.2)$$

The ratio between gap opening penalty o and gap extension penalty e affects the gap penalty cost and hence the optimal alignment between sequences. The size of gaps is more important when o is smaller than e, while it is less important when o is larger than e. The linear gap penalty is a special case of

affine gap penalty when the ratio is equal to 1. Biologically, however, and as stated above, given a fixed number of gaps, an alignment with a small number of long gaps has more likelihood than an alignment with a large number of short gaps. To reflect this, the initiation of gaps is made more expensive than the extension of existing gaps. Taking the example of two biological sequences, *D {M, A, T, T, A, C}* and *Q {M, A, T, C}*, under a linear gap penalty model, the final alignments can be one of the two alternatives shown in Figure 3.1. Under affine gap penalty model, however, the most optimal alignment is given in Figure 3.1.b if opening a gap is more expensive than extending a gap. Such alignment is more likely biologically.

```
M A T T A C      M A T T A C
M A - T - C      M A T - - C
    (a)              (b)
```

Figure 3.1: Two optimal alignments under different gap penalty model

Gotoh [14] presented a local sequence alignment algorithm which permits the construction of local optimal alignment between two sequences in *O(MN)* memory space, where *M* and *N* are the length of sequences, respectively. Since the requirement of memory space grows quadratically with the length of sequences, the applicability of the algorithm is not scalable in practice as the number of alignments grows exponentially. Below, another sequence alignment algorithm, the Smith-Waterman algorithm [15] is present, by which an alignment cost in *O(N)* memory space can be got.

The Smith-Waterman algorithm is a dynamic programming algorithm which aims to find the best local alignment between two biological sequences. The optimal local alignment obtained by the algorithm is achieved in two stages. Firstly, an alignment matrix is calculated based on the correlation between the two sequence characters (e.g., protein amino acids, DNA base pairs). The

optimal local alignment is found by finding the maximum element in the alignment matrix, which assigns a score to the degree of similarity between the two sequences, and tracing back the alignment matrix until a zero element is found. Since the second stage needs large amount of memory space for real world biological sequences, the calculation of alignment matrix to get the score in the first stage is commonly performed only. The trackback is performed for few sequences with the highest alignment scores.

More specifically, let D denote a database sequence of length M: $D_1D_2 \ldots D_m$ and let Q denote a query sequence of length N: $Q_1Q_2 \ldots Q_n$. Define

- $H_{i,j}$ = cost of a substitution between D_i and Q_j.
- $E_{i,j}$ = cost of a conversion of D to Q that deletes Q_j.
- $F_{i,j}$ = cost of a conversion of D to Q that deletes D_i.

The values of $H_{i,j}$, $E_{i,j}$ and $F_{i,j}$ are defined as 0 if $i < 1$ or $j < 1$, otherwise, the value of H satisfies the following relations:

$$H_{i,j} = MAX \begin{cases} E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + W(D_i, Q_j) \\ 0 \end{cases} \quad (3.3)$$

The value of $E_{i,j}$ satisfies Equation 3.4:

$$E_{i,j} = MAX \begin{cases} E_{i,j-1} - e \\ H_{i,j-1} - o - e \end{cases} \quad (3.4)$$

The value of $F_{i,j}$ satisfies Equation 3.5:

$$F_{i,j} = MAX \begin{cases} F_{i-1,j} - e \\ H_{i-1,j} - o - e \end{cases} \quad (3.5)$$

Where o and e are the gap penalty value for opening a gap and extending a gap respectively, and $W(D_i, Q_j)$ stands for the evolution rate between two characters from sequence D and Q based on a substitution matrix W.

From Equation 3.3, it is clear that the score of $H_{i,j}$ depends on the values of its upper neighbour $H_{i-1,j}$, left neighbour $H_{i,j-1}$ and upper-left neighbour $H_{i-1,j-1}$ as illustrated in Figure 3.2. Therefore, to find the best alignment, there are three alternatives:

- An alignment between D_i and Q_j: In this case, the new score $H_{i,j}$ is $H_{i-1,j-1}+W$.
- An alignment between D_i and a gap in Q: In this case, the new score $H_{i,j}$ is the maximum between $H_{i,j-1} - o - e$ and $E_{i,j-1} - e$. If $H_{i,j-1}$ is big enough to open a gap, $E_{i,j}$ is then updated to $H_{i,j-1} - o - e$, which is the new score of $H_{i,j}$, otherwise, $E_{i,j}$ is updated to $E_{i,j-1} - e$, so does $H_{i,j}$.
- An alignment between a gap in D and Q_j: In this case, the new score $H_{i,j}$ is the maximum between $H_{i-1,j} - o - e$ and $F_{i-1,j} - e$, If the $H_{i-1,j}$ is big enough to open a gap, $F_{i,j}$ is then updated to the $H_{i-1,j} - o - e$, which is the new score of $H_{i,j}$, otherwise, $F_{i,j}$ is updated to $F_{i-1,j} - e$, so does $H_{i,j}$.

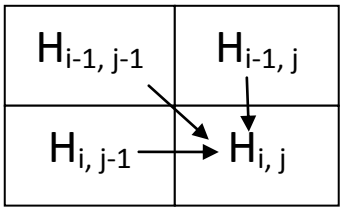


Figure 3.2: Data dependency of the Smith-Waterman dynamic programming algorithm

3.3 A GPU-based Smith-Waterman Algorithm Design and Implementation

Before reporting the details of the novel technique, the two main parallelization strategies previously adopted for the GPU acceleration of the Smith-Waterman algorithm are first described. Afterwards, the improved strategy which solves the problem of query size limitation reported previously is then illustrated.

3.3.1 Intra-task parallelization strategy

Equation 3.3 indicates that the computation of matrix cell $H_{i,j}$ just depends on the values of its upper neighbour $H_{i,j-1}$, left neighbour $H_{i-1,j}$ and its left-upper neighbour $H_{i-1,j-1}$, which means that the calculation of cells within the same anti diagonal of the alignment matrix can be done in parallel. Obviously, for the computation of cells on the k-th anti diagonal, we need to record the cells on the (k-1)-th anti diagonal and the cells on the (k-2)-th anti diagonal. For a query sequence of length *n* and a database subject sequence of length *m*, there are *m+n-1* anti diagonals in total, which need to be computed serially. Instead of storing all matrix cells, we just need to allocate memory for the storage of two consecutive anti diagonals. As illustrated in Figure 3.3, the computation of cells on the i-th row and the k-th anti diagonal (i>0) depends on the values of its left neighbour and upper neighbour on the (k-1)-th anti diagonal which are stored in *shared[i-1]* and *shared[i]* respectively. Moreover, *shared[i]* will be updated when the new *H* value is computed for the computation of the cells in the i-th row. Therefore, we use a register variable for each thread to store the cells on the (k-2)-th anti diagonal and shared memory for the cells on the (k-1)-th anti diagonal. After computing cells on the k-th anti diagonal, we use the cells on the (k-1)-th anti diagonal to update the value of registers and the cells on the current k-th anti diagonal to update the value of shared memory for the computation of all cells on the (k+1)-th anti diagonal. Another register variable defined in the kernel for each thread is used to store and update the highest score of each row. Overall, one block of threads takes charge of computing one matrix and each thread contained within it takes charge of the computation of its row, while multiple thread blocks calculate several alignment matrices in parallel. However, due to the limitation of the maximum number of threads allocated to each block, which is 512 for Geforce 8800 GTX, a problem occurs when the number of residues in a query sequence is bigger than the maximum number of threads defined. Considering this limitation, Munekawa et al. [8] referred to the built-in variable char4 - vector type derived from the basic

character type, it is a structure having 4 accessible components through the fields x, y, z and w respectively [9], for each block to expand the maximum query length possible to 2048 (=512*4) in theory. Nonetheless, the problem still remains for longer query sequences.

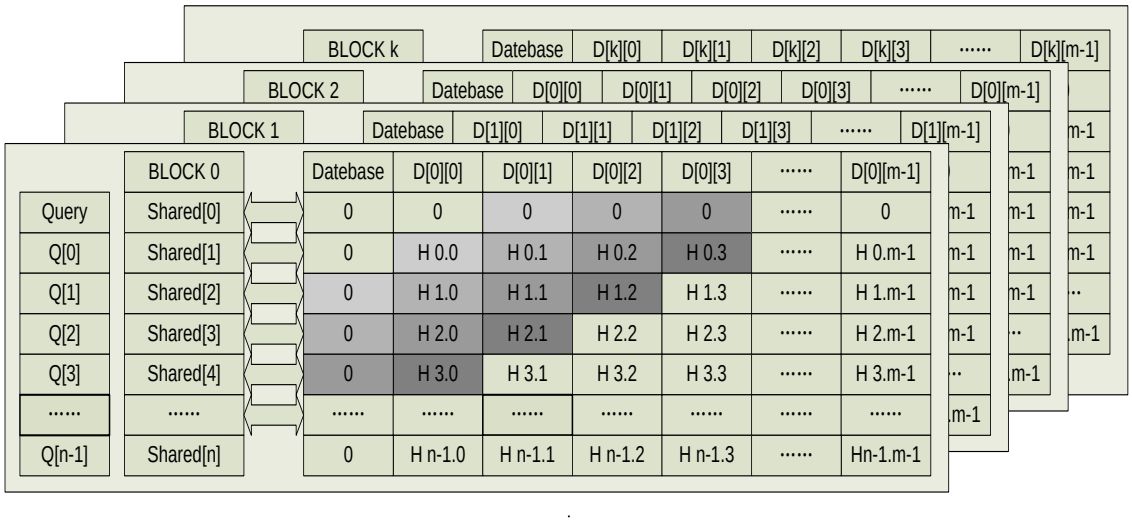


Figure 3.3: Parallel implementation of the alignment matrix computation for k pairs of sequences - the equally shaded parts stand for anti diagonal cells that can be computed in parallel.

3.3.2 Inter-task parallelization strategy

Manavski et al. [7] proposed another different parallelization strategy approach. In it, one single thread is allocated to do the computation of an entire alignment matrix involved by a pair of sequences. For an individual alignment matrix, the computation is processed serially column by column as illustrated in Figure 3.4, while each column is calculated from the top cell to the bottom cell. Since the computation of each column only depends on the previous one, a small amount of memory space is needed, which depends on the length of the query sequence. Though the computation of cells of each matrix is serial for each thread, massive parallel computation can be achieved by using multiple parallel threads to calculate alignment matrices involved by one query sequence and a large amount of subject sequences. In this implementation, however, the amount of parallelism is restricted by the size of

the local memory shared by all parallel threads, which is used to store and load temporary data necessary for the calculation of alignment matrix cells. For different numbers of parallel threads, the available local memory allocated for each thread is not fixed, which will be decreased if more threads are running at the same time. As a result, this method also suffers from a limitation in the maximum possible size of query length to be processed. In next sub section, we will present our proposed method for solving this problem.

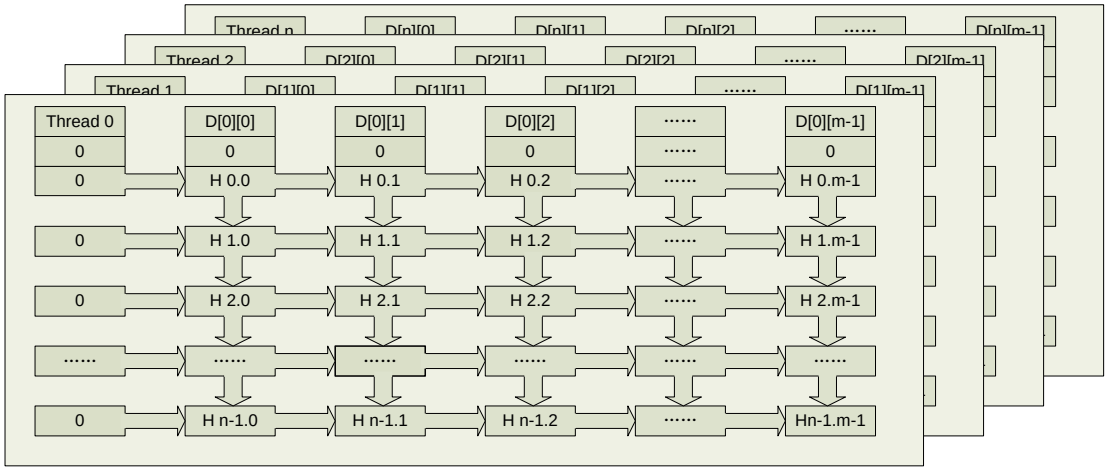


Figure 3.4: Parallel implementation of the alignment matrix computation – one single thread computes a complete alignment matrix of two sequences. Cells in each matrix are computed column by column using local memory as a buffer for temporary data.

3.3.3 The proposed thread iteration strategy

The parallelization strategy harnessed in our design is similar to the approach performed by Munekawa et al. [8], the so called Intra-task parallelization, as we allocate a block of threads to calculate a single alignment matrix. However, in our implementation we separate the computation of single alignment matrix into multiple sub-matrices with a certain number of threads, commensurate with the maximum number of threads and the maximum amount of memory available. Once the batch of threads allocated completes the calculation of a sub-matrix, the final thread in the batch records the elements in the row which it is in charge of and stores them into shared memory or global memory

depending on the size of database subject sequence, ready for the calculation of the next sub-matrix. The first thread in the next batch loads this data as initial data for the subsequent sub-matrix calculation. This operation continues in turn until the end of the entire alignment matrix calculation. This process is illustrated in Figure 3.5 where the final thread in each batch (thread *n*) stores the cells of its row. Afterwards, the first thread in the batch loads these values as initial data for the computation of the first row in the next alignment sub-matrix. It is worth mentioning here that all alignment matrix calculations are done purely on GPU. The host processor only needs to allocate memory on the GPU device and predefines the relevant database sequences offset which guarantees that each block operates on the right section of the database sequence before launching the GPU kernel. Since GeForce 8800GTX can host 24 warps running in each Stream Multiprocessor (SM), thus each SM can have up to 768 (24×32) parallel threads running at the same time [9]. This amount can be split into batches of threads (blocks), where each block computes one alignment matrix. For example, we can split the overall number of threads into

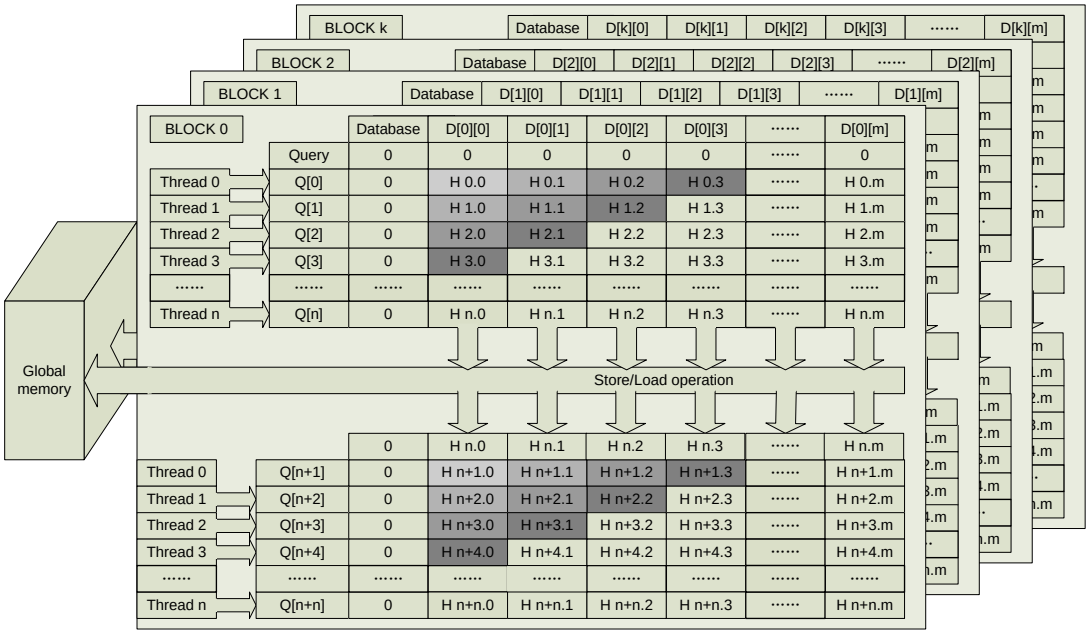


Figure 3.5: The proposed thread reuse strategy, store and load operations are performed by the final thread and the first thread in each thread batch.

8 blocks of 96 threads, with maximum 10 registers (8192/768) allocated to each thread and each block could use up to 2 KB of shared memory. If we use 11 registers for the kernel, the total number of registers is changed to 768×11=8448, which is over the maximum number of registers (8192) defined for each SM, thus the active number of blocks will be decreased to 7. Global memory will be used if this amount of allocated shared memory space is not enough for any database subject sequence. Note here that if the length of the database subject sequence is smaller than the number of threads in the block, additional waiting time should be added for the threads in the batch to finish their computations. This is illustrated in Figure 3.6. If thread 0 has already completed its row calculation, while thread *n* has not completed yet or has not even started its row, then thread 0 of the next block would have to wait for thread *n* of the previous block to complete its task before obtaining its initial data for the next batch of processing i.e. row *n+1*. The waiting time is proportional to the

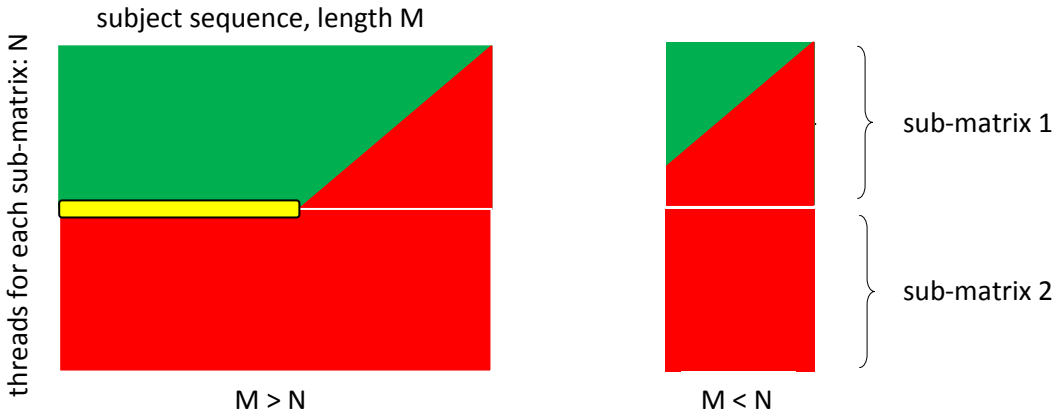


Figure 3.6: Time delay between each sub-matrix. The green colour stands for the cells computed, the red colour stands for the cells not computed and the yellow colour denotes the cells that will be used in the computation of next sub-matrix.

number of threads minus the length of database subject sequence if the length of database subject sequence is smaller than the number of threads, otherwise, it is 0. In the proposed implementation, constant cache is used to store the commonly used constant parameters (e.g., query sequence, gap penalties) in order to decrease access time while shared memory is preferred to store the substitution matrix as it has larger bandwidth. In addition, global memory is

used to store the database sequence as the size of the latter can be in the hundreds of megabytes. Moreover, texture memory is used to shade database sequences. Since H and F in Equation 3.3 need to be shared among threads, Figure 3.7 illustrates four definitions of them and presents whether they lead to bank conflicts. Note that the variable type in (c) is not suitable for long sequences because it only can store small scores (<256). The bottleneck of performance in the implementation is the operation of storing intermediate H and F by the last thread and the load operation by the first thread in each batch, because the latency between SP registers and global memory is much larger than the one between registers and shared memory. No matter how fast other threads execute the kernel, they have to wait for the point where all threads are synchronized. Obviously, this only occurs when the length of the database subject sequence is longer than the allocated space in shared memory. Therefore, our acceleration strategy mainly focuses on the efficient allocation of resources to each block to make use of the maximum available parallelism. This can be achieved through setting the number of threads in each block. Since each SM has 8192 registers and can keep at most 768 threads running at the same time, for a query sequence of length 512, if we use 1 block of 512

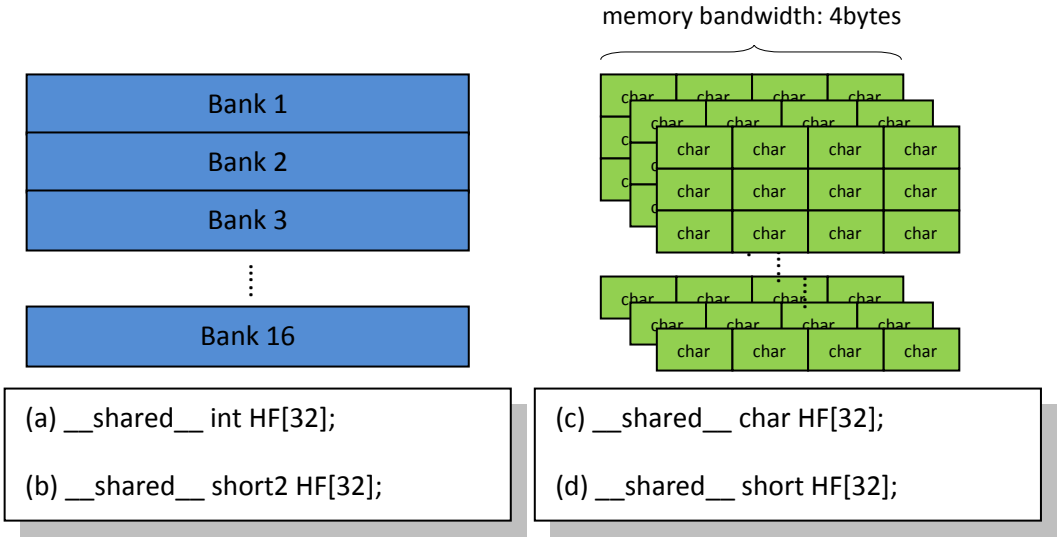


Figure 3.7: shared memory architecture and examples of shared memory definition. The definitions in (a) and (b) do not lead to bank conflicts as threads in the same half-warp access different banks. The definitions in (c) and (d) lead to bank conflicts as some threads within half-warp sit in the same bank.

threads, 16 registers can be used for each thread. In this case, only one sequence alignment can be computed in each SM. If we use 8 blocks of 64 threads, also 16 registers can be allocated to each thread, but the number of sequence alignments which can be processed at the same time becomes 8. Rather than adopting the simple method used by Munekawa et al. [5] which utilized the full memory resource for each block, we flexibly allocate resources through setting the number of threads in each block, with no limitation on the overall length of the query sequence. Table 3.1 presents the execution time of the Smith-Waterman algorithm on GPU using proposed technique, with different numbers of threads per block. For a query sequence of length 1023, 64 threads per block lead to the best performance. Figure 3.8 presents the pseudo code for the proposed kernel. According to the length of query sequence and the amount of threads, the iteration rate of threads is firstly computed in line 3. After that, we compute the additional waiting time for the purpose of insuring the proper initial data for the computation of the next iteration in line 5. Each thread reads the corresponding residues of the query sequence in line 7 by its unique index, e.g. thread ID plus an offset value. Note that there is a stringent

```
QueSeq[QueSize]: Query sequence;
DataSeq[DbSize][DbNum]: Database sequence;
BLOSUM[25][25]:Substitution matrix;
bid: block ID; tid: thread ID;
TLQ:number of query sequence loads;
TLD: number of database sequence loads;
TPB: the number of threads per block;
1. __shared__ int anti_1[TPB + 1];
2. __shared__ int anti_2[TPB + 1];
3. TLQ := QueSize/TPB;
4. TLD := DataSeq[DbSize][bid];
5. wait := TLD – TPB;
6. for QueIndex := 0 to (TLQ-1) do
7.        que := QueSeq[index * TPB + tid];
8.        dia := 0;
9.        anti_1[tid] := 0;
10.       anti_2[tid] := 0;
11.       for DataIndex := 0 to (TLD/2 + wait)
do
12. key := DataIndex – tid;
13. if 0 <= key <= TLD/2
14.       load initial H
15.       data := text1Dfetch[key];
16.       score := blosum[que][data.x];
17.       H := max (0, dia + w, anti_1[tid]-P, H-P);
18.       score := max(score, H);
19.       dia := anti_1[tid];
20.       anti_1[tid + 1] := H;
21.       score := blosum[que][data.y];
22.       H: = max (0, dia + w, anti_2[tid]-P, H-P);
23        score := max(score, H);
24.       dia := anti_1[tid];
25.       anti_2[tid + 1] := H;
21.       store H;
22.endif
23.synchronize threads;
24 output scores into global memory;
25 endfor
```

Figure 3.8: Pseudo code of Intra-task parallelization approach based implementation

Table 3.1: Performance comparison among 64, 128 and 256 threads with all query sequences run against the SWISS-PROT database

Query length	64 threads Time(sec)	128 threads Time(sec)	256 threads Time(sec)
63	2.13	3.1	6.18
127	6.11	4.16	7.15
191	9.28	11.94	8.3
255	12.45	12.93	9.63
511	25.12	26.35	29.17
1023	50.4	53.1	57.8

requirement for the computation order among threads, which is controlled by the variable *key* in line 12. *key* smaller than 0 indicates that the thread needs to wait. When key is equal to 0, it starts fetching residues of database sequences to perform the actual computations of alignment matrix. We harness vector variable *data* in line 15, so that the maximum fetch time can be decreased from *TLD* to *TLD*/2. When *key* > *TLD*/2, it indicates that the thread has already completed the calculation of the final residue of database sequence and there is no need to do any extra operation in this iteration, as shown in line 13. In the final iteration, some threads may not have any pending tasks, in which case they just wait for the synchronization operation in line 23. Lines 16-18 show the computation of the alignment values and the update of the maximum score value. Afterwards, new anti diagonals are updated by modifying variable *dia* in line 19 and the corresponding positions in shared memory in line 20. A similar procedure is performed in lines 21-25 but with different database sequence residues. In order to make the value of residues sit in the right domain, a simple minus operation (characters – '*A*') is implemented before we pass them to the substitution matrix. Apart from the duty of computation, the final thread (*tid* = *TPB*-1) stores *H* for the next iteration in line 21. Load and store operations are added at the start and the end of the inner loop. Finally, every thread copies the highest score of all rows it is in charge of to global memory in line 24. Each block executes the code with the same query sequence and different database sequences; output is the maximum score of each row in

each matrix. tid and bid represent the ID of each individual thread and block respectively. P stands for gap penalty value. We choose linear gap model as described in section 3.2 to perform the comparison among 64, 128 and 256 threads for the sake of simplicity and the comparison with Liu et al [6] for consistency and affine gap model for others. The substitution matrix is stored in shared memory.

3.3.4 Results and Evaluation

In this section, the experimental results of our Smith-Waterman GPU implementation compared to the state-of-the-art are presented. In the proposed implementations, a Mac Pro desktop computer running Ubuntu 8.10 32-bit Linux operation system is used, with an NVIDIA GeForce 8800 GTX GPU installed. The experiments reported query sequences of lengths ranging from 63 to 4095 amino acids. All query sequences run against the Swiss-Prot protein sequence database [16], which is approximately 180MB in size, and contains 399,749 sequence entries with a total of 144,041,553 amino acids. The cell updates per second (CUPS) is commonly used for measuring the Smith-Waterman execution performance as it normalises in terms of database and query sequence sizes. Given a query sequence Q and a sequence database D, the Mega CUPS (MCUPS) is defined as:

$$MCUPS = \frac{|Q| \times |D|}{t \times 10^6} \quad (3.6)$$

where $|Q|$ denotes the length of the query sequence, $|D|$ denotes the total length of the database sequences, and t denotes the elapsed time in seconds. From Equation 3.6, we can find the value of MCUPs is inversely proportional to the elapsed time, which means bigger MCUPS, better performance. The version of the Swiss-Prot database used in Liu et al [6] is release 46.3, March 2005, which contains 176,469 sequence entries, with an average length of 361 amino acids. The GPU type used in their implementations is an NVIDIA GeForce 7800GTX GPU. For the sake of simplicity, they used a simple

substitution matrix: +2 if the characters are identical and -1 otherwise. In the implementation, the substitution scoring strategy was made by using the more accurate BLOSUM50 matrix. Moreover, the performance with the simple version of substitution matrix for the purpose of fair comparison is tested, the results of which are shown in Table 3.2. The evaluation of our implementation on the GeForce 8800GTX GPU shows a speedup factor from 4x to 20x, the factor can be calculated by dividing the MCUPS between two approaches. However, the two implementations were made on GPUs of different generations. In order to allow for a fairer comparison, we compared our implementation with more recent GPU implementations from Munekawa et al. [8] in Table 3.3 and Manavski et al. [7] in Table 3.4. Since query sequences with length 1023 and 2047 were not provided in their report, hence we do not include these two values in the tables. Note that the number of threads defined in each block is 64 in our proposed method.

Table 3.2: Performance comparison between Liu's method and the proposed method, both are using a simple substitution matrix

Query length	Execution time (sec)		Throughput(MCUPS)	
	Proposed	Liu's	Proposed	Liu's
63	2.13	19.5	4059	196
127	4.16	25	4189	308
255	9.63	36.3	3634	427
511	25.12	59.2	2792	524
1023	50.4	105.1	2786	591
2047	101.12	197.9	2778	628
4095	202	383.1	2782	649

Table 3.3: Performance comparison between Munekawa's method and the proposed method

Query length	Execution time (sec)		Throughput (MCUPS)	
	Proposed	Munekawa's	Proposed	Munekawa's
63	2.13	2.96	4059	1838
127	4.16	3.38	4189	3244
191	7.36	3.98	3561	4121
255	9.63	4.66	3634	4725
511	25.12	8.19	2792	5388
4095	202	N/A	2782	N/A

Table 3.4 presents comparative implementation results between the proposed implementation and Munekawa et al.'s which targeted NVIDIA's GeForce 8800GTX GPU. Here, we can find that for small size query sequences, our implementation performs better, but as query sequence length increases, the performance of our implementation decreases. This is because of the overheads associated with storing and loading intermediate data between computation batches when the query sequence length is greater than the number of threads, as explained in the previous section. Nonetheless, we notice that Munekawa et al.'s implementation cannot cope with query sequences longer than 2048, whereas our implementation can cope with any query sequence length. This is a major advantage of our method which makes it completely usable in real world bioinformatics applications. Note that the difference between throughput ratios and execution time ratios in Table 3.3 is due to the use of different versions of the Swiss-Prot database, i.e. with different sizes. Table 3.4 presents comparative results with another recent GPU implementation of the Smith-Waterman algorithm, from Manavski et al. targeted on an NVIDIA GeForce 8800GTX GPU which was explained in detail in section 3.1.2 above. This implementation has speed advantages over our proposed one, due to the higher amount of parallelism allowed by computing several alignment matrices in parallel coupled with a simpler synchronization mechanism allowed by the fact that only one thread is associated to each

Table 3.4: Performance comparison between Manavski's method and the proposed method

Query length	Execution time (sec)		Throughput (MCUPS)	
	Proposed	Manavski's	Proposed	Manavski's
63	8	2.98	1080	1849
127	15.5	5.88	1124	1889
255	35	12.31	1000	1811
511	81.8	24.89	857	1795
4095	633.6	N/A	885	N/A

alignment matrix calculation. The superiority becomes more obvious when aligning longer query sequences. However, this implementation suffers from

the local memory size bottleneck which limits the length of query sequence up to 2048. The proposed implementation on the other hand does not suffer from any such limitations, and thus can be fully usable in real world bioinformatics application. Nonetheless, the inter-task parallelization approach still has very important advantages. an improved strategy on Manavski et al.'s implementation to conquer the problem and discuss the difference and choice of inter-task and intra-task approaches will be presented in section 3.4. In addition to the above, the performance of our GPU implementation is compared with a widely used CPU-based implementation of the Smith-Waterman algorithm, namely SSEARCH from the FASTA set of programs [4]. Table 3.5 presents comparative results of the proposed GPU implementation with an equivalent SSEARCH (version 35.04) implementation on a Pentium4 3.4GHz desktop computer running Windows XP OS. This shows our GPU implementation outperforms the equivalent CPU implementation by up to 15x.

Table 3.5: Performance comparison between SSEARCH and our proposed method

Query length	Execution time (sec)		Throughput (MCUPS)	
	Proposed	SSEARCH	Proposed	SSEARCH
63	8	125	1080	70
127	15.5	210	1124	83
255	35	424	1000	83
361	56.4	536	880	92
511	81.8	779	857	90
2640	424	4053	854	89
4095	658	5808	854	96

3.4 Inter-task parallelization vs. Intra-task parallelization

Two task parallelization strategies on GPU and the proposed implementation have been presented in the above sub-sections. The inter-task approach outperforms the intra-task approach may be deduced from the above results. This section firstly presents our design and implementation of the Smith-Waterman algorithm based on the inter-task parallelization technique. It then evaluates the performance of the inter-task parallelization and the intra-task

parallelization approaches and discusses the choice between the two alternatives. Figure 3.9 illustrates the computation method adopted for our inter-task parallelisation approach. We harnessed a divide and conquer approach as described in section 3.3.3 to complete the computation. Since the GPU device is capable of reading and writing 128-bit words between global memory and registers, we pre-process the database and query sequences and pack them into clusters with 4 residues in each one. Suppose there is a database sequence of length *N* and a query sequence of length *M*, the access demands of *H* or *E* value between global memory and registers can be decreased from *M*(N-1)* to *(M/4)*(N/4 – 1)*. For instance, if both *M* and *N* comprise 4 residues, the access occurrence is 0, as intermediate values are stored in build-in variable *int4*. If *N* comprises 8 residues, it is then packed into 2 clusters, and the access occurrence becomes to 1. In order to make the most efficient use of the bandwidth, database sequences must be well aligned as shown in Figure 3.10a so that threads within the same half-warp can achieve

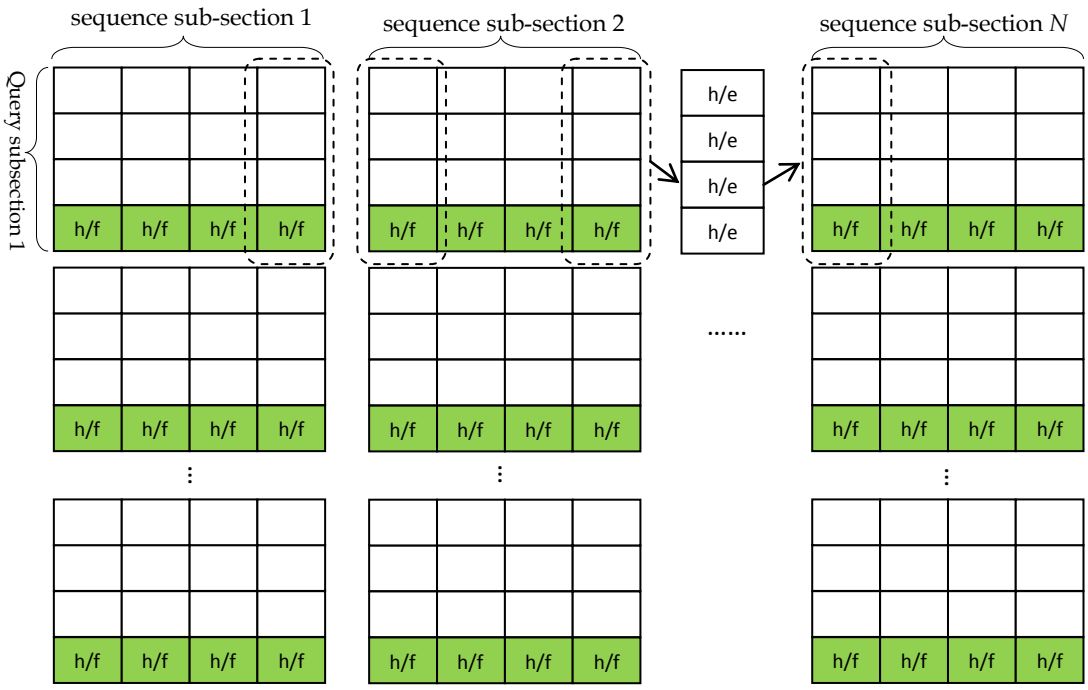


Figure 3.9: The computation architecture of the alignment matrix by using inter-task parallelization approach. Each thread calculates the matrix involved by same query sequence, but different database sequence.

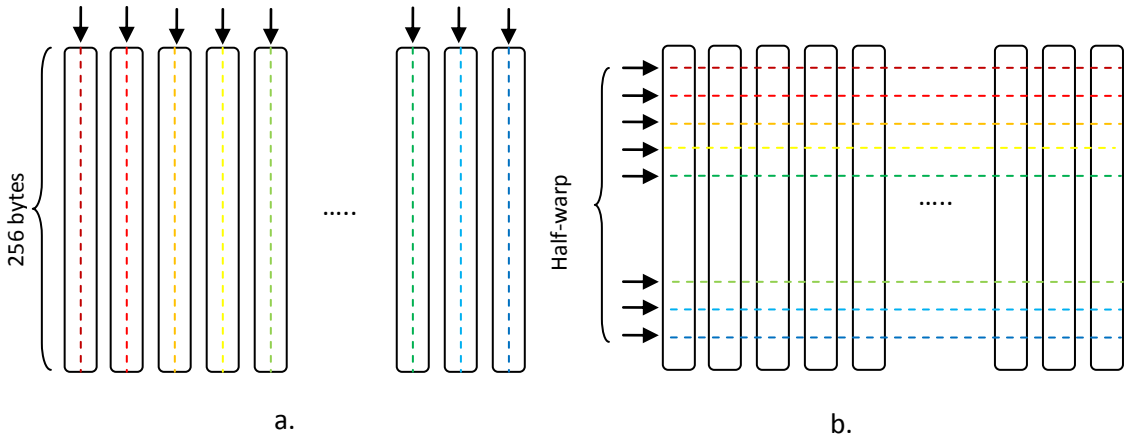


Figure 3.10 Two different sequence memory patterns. (a) Non-coalesced access: the access is serialized to 16 readings. (b) Coalesced access: only one reading is performed.

the right access pattern, the so called coalesced access. More precisely, thread number *t* within the same half-warp should access *HalfWarpStartingAddress[t]*, Where *HalfWarpStartingAddress* should be aligned to 16*sizeof(type) bytes, the size of variable accessed is equal to 4, 8 or 16 bytes. Note that any global memory allocations need to be aligned to at least 64 bytes to satisfy the memory alignment constraint, as one improper global memory can lead all other arrays which are allocated afterwards to be misaligned. Figure 3.11 shows the implementation pseudo code of the computation architecture illustrated in Figure 3.9. As discussed in section 3.3.2, Manavski et al [7] selected local memory to store intermediate data in their implementation. Since there are totally *16k* bytes local memory space for each thread in the GPU they used, therefore the maximum length of query sequence was 16*1024/(*H*&*F*) = 16*1024/8 = 2048. In order to overcome this problem, we use global memory instead. Moreover, we harnessed the divide and conquer strategy so that some intermediate data can be temporarily stored in shared memory. Table 3.6 presents the performance of the proposed method. Figures 3.12 and 3.13 illustrate the performance comparisons between the two parallelization strategies for the implementation of the Smith-Waterman algorithm, where

```
int4        h, e;
int         q_x, q_y, q_z, q_w;
int         d_x, d_y, d_z, d_w;
__shared__ int      s_h[cluster];
__shared__ int      s_f[cluster];

1.    h := g_h[q->index]; e := g_e[q->index];
1.    For  ch := d_x to d_w do;
1.          e := max(e.x – c_ex, h.x – c_oe);
2.          f := max(f - c_ex, r – c_oe);
3.          r := max(blosum[q_x][ch], e, f, 0);
4.          s := max(r, s);
5.          d := h.x;
6.          e.x := e; h.x := r;

7.          e := max(e.y – c_ex, h.y – c_oe);
8.          f := max(f - c_ex, r – c_oe);
9.          r := max(blosum[q_y][ch], e, f, 0);
10.         s := max(r, s);
11.         d := h.y;
12.         e.y := e; h.y := r;

13.         e := max(e.z – c_ex, h.z – c_oe);
14.         f := max(f - c_ex, r – c_oe);
15.         r := max(blosum[q_z][ch], e, f, 0);
16.         s := max(r, s);
17.         d := h.z;
18.         e.z := e; h.z := r;

19.         e := max(e.w – c_ex, h.w – c_oe);
20.         f := max(f - c_ex, r – c_oe);
21.         h := max(blosum[q_w][ch], e, f, 0);
22.         s := max(r, s);
23.         d := s_h[ch->index];
24.         e.w := e; h.w := h;

25.         s_h[ch->index] := r;
26.         s_f[ch->index] :=  f;

27.         g_h[q->index] := h;
28.         g_e[q->index] :=  e;
29. endfor
```

Figure 3.11: The pseudo code of the most inner loop of the proposed inter-task parallelization, q_x-w and d_x-w denote the relevant query residues and database residues respectively.

Table 3.6: The performance of the proposed inter-task parallelization method, the target database comprises 230152 sequences and 84480541 residues.

Name of query sequence	Length	Time (sec)
O19927	32	1.20
A4T9V0	64	2.26
Q2IJ63	128	4.36
Q96B36	256	9.12
QJLB7	512	18.56
P08715	1024	37.34
Q8IYD8	2048	76.82
QO6277	4095	157.45

both Intra-task parallelization and Inter-task parallelization techniques have been performed to implement the Smith-Waterman algorithm on GPU for different sequence lengths and database sizes. The length of query sequences is around 860 for long sequence databases and 270 for short sequence databases. Compared with the inter-task parallelization approach, the Intra-task parallelization approach achieves more efficient bandwidth and smaller size of device memory requirements. It is normally used for relatively small sets of

sequences where shared cache is commonly exploited for faster data accessing. Inter-task parallelization approach is normally used for relatively large sets of sequences, as device memory access latency can be hidden by many independent threads. However, this process needs much more device memory resource. The Figure 3.12 and 3.13 clearly show that intra-task parallelization approach outperforms the Inter-task parallelization approach for relatively small databases with short query sequences and that inter-task parallelization performs better for relatively large databases with longer query sequences. On the one hand, insufficient tasks result in thread load imbalance, to fulfil the hardware of GPUs and the maximum number of threads, the intra-task parallelization strategy is more advisable. Adversely, longer tasks are involved in inter-task parallelization, as the runtime of each thread is relatively longer. On the other hand, shared memory is employed continually in intra-task parallelization strategy, which directly impacts the performance as it is much faster than the local or global memory.

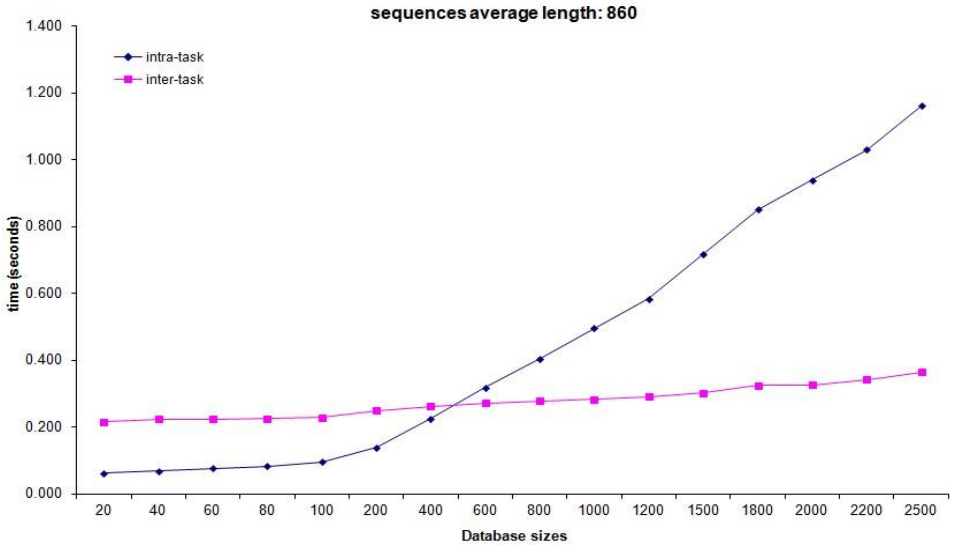


Figure 3.12: Performance comparison between inter-task and intra-task parallelization strategies for 16 groups of long sequences

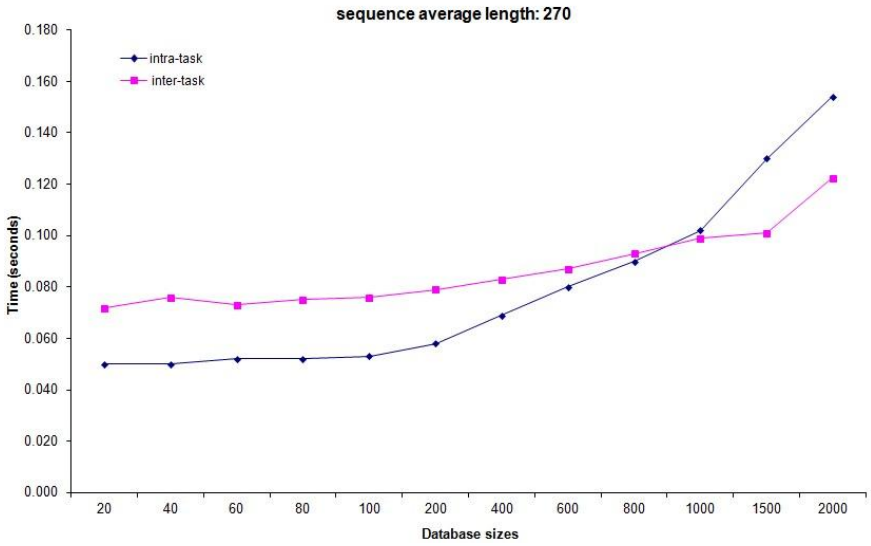


Figure 3.13: Performance comparison between inter-task and intra-task parallelization strategies for 16 groups of short sequences

3.5 Conclusions

In this chapter, a novel technique for the implementation of the Smith-Waterman algorithm on CUDA-compatible GPUs was presented. The strategy solves the query length limitation reported in previous GPU implementations of the Smith-Waterman algorithm, as it can cope with any length of query or subject sequence. Central to this strategy is a divide and conquer approach to alignment matrix calculation in which the size of sub-matrix calculation is dictated by the available computing and memory resources in the GPU hardware, with thread iteration across all sub-matrix calculations. This however comes at a speed overhead due the storing and loading of temporary intermediate data in the global memory. Despite this speed penalty, the proposed GPU implementation still outperforms an equivalent CPU-based implementation by up to 15x. Moreover, a comparison between two main parallelization techniques, namely inter-task parallelization and intra-task parallelization was performed, which showed a trade-off between the two strategies in that Intra-task parallelization works better for relatively small databases with smaller query sequences and Inter-task parallelization performs better for relatively large databases with longer query sequences.

3.6 References

[1] Hasan L., Al-Ars Z. and Vassiliadis S.: Hardware acceleration of sequence alignment algorithms-an overview. International Conference on DTIS, pp.92-97, 2-5 Sept. 2007.

[2] Pearson W.R., Lipman D.J.: Improved tools for biological sequence comparison.Proc Natl Acad Sci USA 1988, 85:2444-2448.

[3] Altschul S.F.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.Nucleic Acids Res 1997, 25:3389-3402.

[4] Pearson W.R.: Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms.Genomics 1991,11:635-650.

[5] Charalambous M., Trancoso P., Stamatakis R.: Initial experiences porting a bioinformatics application to a graphics processor.In Proceedings of the 10th Panhellenic Conference on informatics 2005. pp.415-425

[6] Liu W.G.: Bio-sequence database scanning on a GPU.In Proceedings of the 20th International Parallel and distributed Processing symposium 2006.

[7] Manavski S.A., Valle G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 2008.9(Suppl 2):S10

[8] Munekawa Y., Ino F., Hagihara K.: Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU. In Proceedings of the 8th IEEE International Conference on BioInformatics and BioEngineering 2008. pp.1-6

[9] "CUDA Programming Guide Version 1.1", retrieved 10th Oct, 2008 http://developer.nvidia.com/cuda/

[10] Kruger J., Westerman R.: Linear algebra operators for gpu implementation of numerical algorithm. ACM Trans. Graph 2003, 22:908-916.

[11] Agarwal P.K., Krishnan S., Mustafa N: Streaming geometric optimization using graphics hardware. In Proceedings of the 11th European Symposium on Algorithms 2003. pp.544-555

[12] Xu F., Muller K.: Ultra-fast 3d filtered back-projection on commodity graphics hardware. In IEEE International Symposium on Biomedical Imaging 2004, pp.571-574.

[13] Farrar M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 2007, 23(2):156–161.

[14] Gotoh O.: An improved algorithm for matching biological sequences. J Molecular Biology 1982, 162(3):705-708

[15] Smith T.F., Waterman M.S.: Identification of common molecular subsequences. J. Molecular Biology 1981, 147:195–197.

[16] Bairoch A., Apweiler R.: The SWISS-PROT protein knowledgebase and its supplement TrEMBL. Nucleic Acid Research, release 56.3, 14-Oct-08.

Chapter 4

Design and Implementation of a CUDA-compatible GPU-based Core for Gapped BLAST Algorithm

4.1 Introduction

Biological sequence alignment algorithms with exhaustive search strategy, such as the Smith-Waterman algorithm [1] and the Needleman-Wunsch algorithm [2], are computationally intensive and expensive. These algorithms cause relatively long execution time on normal commercial computers when aligning a sequence against a database containing millions of subject sequences. In order to overcome the shortages caused by the complexity of exhaustive dynamic programming algorithms, many attempts have been made to produce faster algorithms. Heuristic methods are developed to speed up the process of finding a satisfactory solution at a lower computational cost. Heuristic algorithms are widely applied in problems, whereby optimal solutions are too prohibitive to achieve, e.g. in Computer science [3], Psychology [4] and Engineering [5].

BLAST [6] and FASTA [7] are two of the best-known heuristic algorithms for biological local sequence alignment. Their basic idea is established on the fact that most sequences in biological databases do not match or few of them have similarities so that some approaches can be applied to exclude many of the unrelated sequences. The BLAST algorithm search heuristically for high-scoring segment pairs (HSPs) in the alignment matrix involved between a pair of sequences (usually a query sequence and a subject sequence from a biological database) before a local extension around these HSPs is performed. Since BLAST takes a heuristic approach to sequence alignment, it achieves less accurate results compared to the gold standard Smith-Waterman algorithm, but runs much faster in return on general purpose processors (GPPs). In general, the speed-up is more than one order of magnitude. For instance,

BLAST (blastp 2.2.21+) requires few seconds for a typical search of the Swiss-prot database on a modern desktop hardware, which is typically 50x faster than SSEARCH [8]. Because of the popularity of BLAST in the Bioinformatics community, this chapter presents the detailed design and implementation of the latest version of BLAST, namely Gapped BLAST with two-hit method [9], on CUDA-compatible GPUs.

The remainder of this chapter is organized as follows. First, essential background on the BLAST algorithm is presented. Then, the details of the proposed multi-threaded parallel design and implementation of the Gapped BLAST with two-hit method algorithm on GPUs are presented. Afterwards, performance comparison between the proposed implementation and the gapped BLAST implementation software (BLASTP 2.2.21+) hosted by the NCBI follows before conclusions are laid out.

4.2 Background

4.2.1 Essentials of the BLAST algorithm

BLAST is a heuristic algorithm which filters dissimilar regions in sequence pairs, as it consists of three heuristic layers, namely seeding, extension and evaluation. Unlike the Smith-Waterman algorithm, BLAST does not explore the entire search space between two sequences. Minimizing the search space is the key to its speed, at the cost of a loss in sensitivity however [10]. This section presents the Gapped BLAST with the two-hit method algorithm. The first step consists in rapidly comparing sequences from database with a given query sequence to identify hits, the score of which is over a particular threshold value (T). The second step performs the ungapped extension between two hits on the same diagonal line, which aims to find high scoring segment pairs (HSPs). The third step performs gapped extension on eligible HSPs after an evaluation by using a modified Needleman-Wunsch algorithm. We present each of these steps in detail below.

Step1. The first step begins by extracting fixed length overlapping sub-sequences (words), from subject sequences. The default length w is equal to 3 for amino acid sequences, while w is equal to 11 for nucleic acid sequences. For instance, suppose w=3 for a protein fraction *MKFVLL*. The words extracted from it are *MKF*, *KFV*, *FVL* and *VLL,* respectively. In general, for a sequence of length m $(m>2)$, the number of words extracted is $(m - w) + 1$. Afterwards, the extracted words are compared with words of the same length from the query sequence one by one using a mutation scoring matrix as illustrated in Figure 4.1. A match between a pair of words is considered as high score if the words are identical or the comparison score is over a particular threshold value. This process is illustrated below with one word from a query sequence aligning three words from a subject sequence, where the scoring strategy is based on the BLOSUM62 matrix.

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	9	-1	-1	-3	0	-3	-3	-3	-4	-3	-3	-3	-3	-1	-1	-1	-1	-2	-2	-2
S	-1	4	1	-1	1	0	1	0	0	0	-1	-1	0	-1	-2	-2	-2	-2	-2	-3
T	-1	1	4	1	-1	1	0	1	0	0	0	-1	0	-1	-2	-2	-2	-2	-2	-3
P	-3	-1	1	7	-1	-2	-1	-1	-1	-1	-2	-2	-1	-2	-3	-3	-2	-4	-3	-4
A	0	1	-1	-1	4	0	-1	-2	-1	-1	-2	-1	-1	-1	-1	-1	-2	-2	-2	-3
G	-3	0	1	-2	0	6	-2	-1	-2	-2	-2	-2	-2	-3	-4	-4	0	-3	-3	-2
N	-3	1	0	-2	-2	0	6	1	0	0	-1	0	0	-2	-3	-3	-3	-3	-2	-4
D	-3	0	1	-1	-2	-1	1	6	2	0	-1	-2	-1	-3	-3	-4	-3	-3	-3	-4
E	-4	0	0	-1	-1	-2	0	2	5	2	0	0	1	-2	-3	-3	-3	-3	-2	-3
Q	-3	0	0	-1	-1	-2	0	0	2	5	0	1	1	0	-3	-2	-2	-3	-1	-2
H	-3	-1	0	-2	-2	-2	1	1	0	0	8	0	-1	-2	-3	-3	-2	-1	2	-2
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5	2	-1	-3	-2	-3	-3	-2	-3
K	-3	0	0	-1	-1	-2	0	-1	1	1	-1	2	5	-1	-3	-2	-3	-3	-2	-3
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5	1	2	-2	0	-1	-1
I	-1	-2	-2	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4	2	1	0	-1	-3
L	-1	-2	-2	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4	3	0	-1	-2
V	-1	-2	-2	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4	-1	-1	-3
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6	3	1
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7	2
W	-2	-3	-3	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11

Figure 4.1: Blosum62 substitution matrix

Query word: M K F

5 + 5 + 6 = 16

word 1: M K F

Query word: M K F

5 + 5 + 3 = 13

word 2: M K Y

Query word: M K F

0 + 5 + -3 = 2

word 3: Q K K

Word 1 is identical with the query word, while word 2 has much similarity with the query word. Their scores are 16 and 13, respectively. Assuming a threshold value is 12, the address of word 1 and 2 in the sequences are recorded as their score values are over the threshold. Word 3 is abandoned, as it scores less than the threshold value. A lookup table (see Figure 4.2) is constructed for the purpose of facilitating the matching of subject sequence words with all the possible matching query sequence words. This lookup table can be used to express all possible words. The size of the table depends on both word size w and the number of alphabet in the biological sequence used. Since there are 25 continuous alphabets from *A* to *Y* in the BLOSSUM62 mutation matrix, with five unused alphabets (*B, J, O, U, X*), each amino acid can be coded as a five-bit unique binary value ($2^5 > 25$). For words with size equal to 3, the address of the lookup table is from *'00000 00000 00000'* to *'11001 11001 11001'*. In the table, the slots of address which are composed by unused alphabets are not used. Overlapping words extracted from the query sequence are compared with all the words presented by their addresses in the lookup table one by one. Three letters in each address can be restored from the address number through the following operations: address>>10, (address>>5)&31, address&31 respectively. Addresses with score values over the specified threshold value are labelled. After the construction of the lookup table, overlapping words from subject sequences are then extracted and transformed into 15-bit binary codes, named the table offsets. These offsets are used to fast

access the exact positions in the lookup table, checking if there are identical or similar words existing and their positions in the query sequence. For example, let *DCT* denote a word from the subject sequence, the accessing address is *'00011 00010 10011'* in lookup table and the offset value can be computed as *('D'-'A')<<10 | ('C'-'A')<<5 | ('T'-'A')*. Suppose the following character is *M*, the next word is then updated to *CTM*. To achieve this, the bits standing for *C* and *T* in the previous word are kept through a bit *AND* operation with binary '*11111 11111*', resulting in a 10-bit binary representation of *CT*, then left shifted by 5 bits, and *OR*ed with *('M'-'A')* to insert the *'M'*. Once a hit is found, the positions in the subject sequence and the query sequence are passed into step2, namely the ungapped extension.

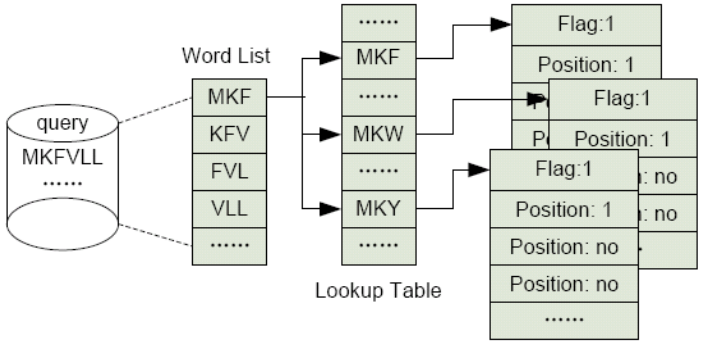


Figure 4.2: The structure of a lookup table

Step2. The original BLAST algorithm extends alignments between the query sequence and subject sequence in both left and right directions for all hits. These extensions are terminated when the accumulated score values of the HSPs fall a certain cut-off depth *X1* below the maximum score values obtained so far (see Figure 4.3). After the alignment terminates, it is trimmed back to the maximum score. In order to save more time, but also keep the same sensitivity as the original BLAST, a new generation of BLAST, namely Gapped BLAST [9], has been proposed. It takes a relatively lower threshold value (*T*) than the original BLAST to guarantee more potential hits to be found. However, only few hits are paired with a second hit on the same diagonal to be extended.

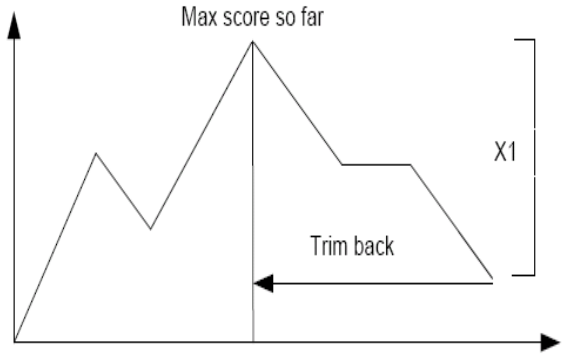


Figure 4.3: Attenuate extension with depth X1

Therefore, the frequency to trigger ungapped extension in Gapped BLAST is less than the original BLAST algorithm. If there are two non-overlapping hits *h1[d1, q1]* and *h2[d2, q2]* on the same diagonal within distance *A=q2-q1* such that *2<A<40*, inward ungapped extension will be triggered. The two hits are combined as a new region by computing the score of residues between them without deletion and insertion i.e. without gaps (see Figure 4.4).

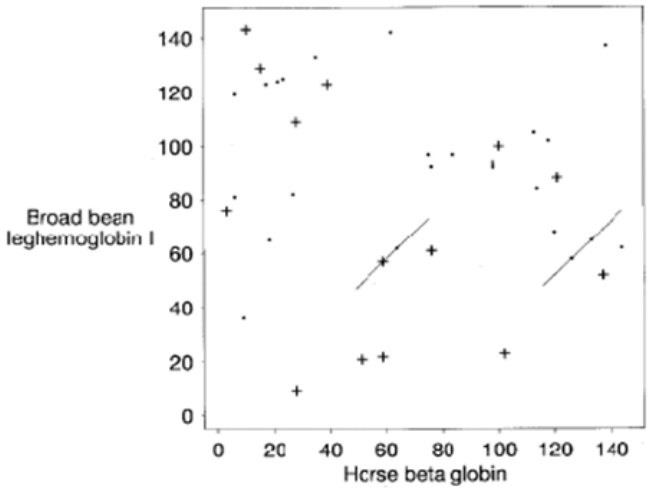


Figure 4.4: Ungapped extension of the two close hits on the same diagonal [9].

Afterwards, the outward ungapped extension starts from the edge of the left and right hand sides of the two hits and terminates when the accumulated total score falls a certain cut-off depth below the best score obtained so far.

Then, the extension is trimmed back to its state with the best score obtained. Finally, if the alignment of ungapped extension has a sufficiently high score *S*, it is then passed into step3, namely gapped extension.

Step3. Gapped BLAST is an advancement of BLAST with the two-hit method. It allows for the insertion of gaps on subject sequence and query sequence. To achieve this, the alignment of ungapped extension is extended both backwards and forwards from the centre of the ungapped alignment (see Figure 4.5).

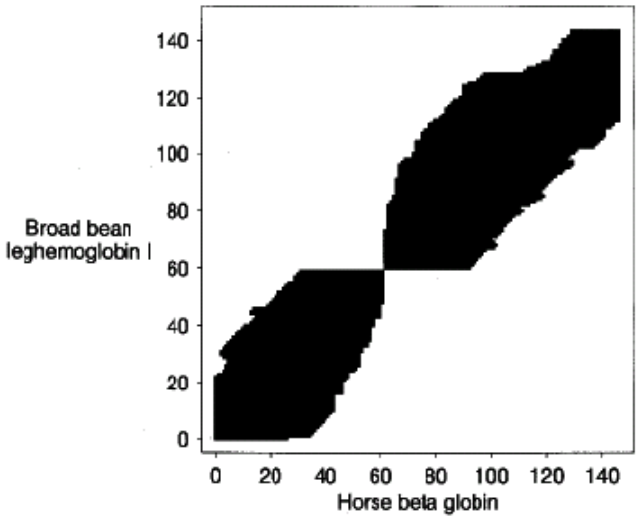


Figure 4.5: Gapped alignment started from the central pair of the ungapped alignment in both directions [9].

The algorithm for gapped alignment utilized in Gapped BLAST is a modified version of the Needleman-Wunsch algorithm where the alignment is pruned when alignment scores (in both directions) fall below a certain cut-off value. In the next sub-section, the essential background of the global alignment algorithm will be presented.

4.2.2 Essentials of the Needleman - Wunsch algorithm

The Needleman-Wunsch algorithm (NW) is a dynamic programming algorithm which aims to search for the optimal global alignment between two biological sequences. Since we presented another similar dynamic programming algorithm – the Smith-Waterman algorithm (SW) in chapter 1, here the difference between these two algorithms is presented. The fundamental principle of the two algorithms is to calculate the score values of a similarity matrix. NW was the first application of dynamic programming to biological sequence alignment. SW algorithm is based on NW algorithm, but instead of searching the entire optimal alignment, it compares segments of all possible lengths, but only chooses the one with maximum similarity. Assuming two sequences $X\ \{ x_1 x_2 \dots x_m \}$ of length M and $Y\ \{ y_1 y_2 \dots y_n \}$ of length N, a dynamic programming score matrix $H(M \times N)$ is constructed where a cell score $H_{i,j}$ represents the alignment score value between $\{ x_1 x_2 \dots x_i \}$ and $\{ y_1 y_2 \dots y_j \}$, the cell score $H_{m,n}$ denotes the similarity between the complete sequences X and Y. The optimal alignment trace path starts from the most bottom right cell $H_{m,n}$ to the most upper left cell $H_{1,1}$. Under linear gap model, the boundary cells of matrix H are set by Equation 4.1:

$$\begin{cases} \mathrm{H(i,j)} = 0 \ \text{ if } \mathrm{i} = 0, \mathrm{j} = 0 \\ \mathrm{H(i,0)} = \ \mathrm{i} \times \text{penalty, where } \mathrm{i} = 1, 2, \dots \dots \mathrm{M} \\ \mathrm{H(0,j)} = \ \mathrm{j} \times \text{penalty, where } \mathrm{j} = 1, 2, \dots \dots \mathrm{N} \end{cases} \quad (4.1)$$

Equation 4.2 is used to compute the score values of each cell in matrix H. The difference with the SW algorithm is that NW algorithm does not set negative score values to 0.

$$H(i,j) = \max \begin{cases} H(i-1,j-1) + s(x_i, y_j) \\ H(i-1,j) + \text{gap penalty} \\ H(i,j-1) + \text{gap penalty} \end{cases} \quad (4.2)$$

Under affine gap model, in addition to the computation of *H*, there are two new matrices *E* and *F* to compute. The boundary cells of matrix *E*, *F* and *H* are set by Equations 4.3 and 4.4, where *o* and *e* denote the open penalty and extending penalty values, respectively.

$$\begin{cases} H(i,0) = i \times e \\ E(i,0) = o + i \times e \end{cases} \text{where } i = 1, 2, \ldots \ldots M; \quad (4.3)$$

$$\begin{cases} H(0,j) = j \times e \\ F(j,0) = o + j \times e \end{cases} \text{where } j = 1, 2, \ldots \ldots N; \quad (4.4)$$

Equations 4.5, 4.6 and 4.7 give the computation formulas of matrix *H*, *E* and *F*, respectively.

$$H(i,j) = \max \begin{cases} H(i-1,j-1) + s(x_i, y_j) \\ E(i,j) \\ F(i,j) \end{cases} \quad (4.5)$$

$$E(i,j) = \max \begin{cases} E(i,j-1) - e \\ H(i,j-1) - o \end{cases} \quad (4.6)$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - e \\ H(i-1,j) - o \end{cases} \quad (4.7)$$

4.3 Design and Implementation of Gapped BLAST with Two-Hit Method on GPU

As the gapped BLAST consists of a series of independent steps, three kernels are designed to parallelize its implementation on GPU, which are the parallelism of the lookup table construction, the two-hit method and the gapped extender respectively. Below the organization of the proposed method is presented in general. Then, the detailed implementation of each kernel is presented.

4.3.1 High Level Application Software

Figure 4.6 shows the organization of the proposed GPU-based core for Gapped BLAST algorithm with two-hit method, which is also the typical architecture for GPU computing.

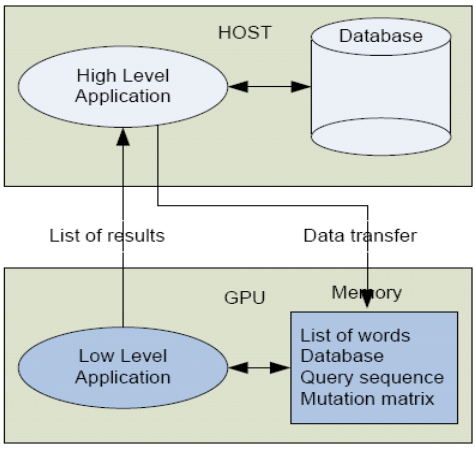


Figure 4.6 Organization of the proposed method

In a host-based high level application, residues in the subject sequence database e.g. Swiss-Prot and query sequence are read into device memory. Since the subject sequence database is relatively large, they are all stored in global memory. Global memory should be aligned to multiples of 256 for fast coalesced memory access. The query sequence and substitution matrix are

used to construct the lookup table. They are stored in constant memory and read into shared memory by each thread block later. Since the number of hits for each address in the lookup table is not fixed and unpredicted, and it is not feasible to dynamically allocate memory after launching GPU kernel, we normally allocate a large size of memory for each address. All hit information will be sent back and compressed on the host application.

4.3.2 The Parallelism of the Lookup Table Construction

As described in sub-section 4.2.1, the word size is equal to 3 for protein sequence and the maximum value of lookup table address is as '11001 11001 11001' in binary and 26,425 in decimal. Assuming there is a protein query sequence of length H, the number of words extracted from it is $H - 1$. The computation complexity is then equal to $O(26{,}452 \times (H\text{-}1))$. The proposed parallelism approach is based on the fact that filling the lookup table can be fully parallelized as each entry is independent of the others. Since the size of the lookup table is fixed, a fixed number of threads is allocated in advance to it, which is achieved by setting *dimBlock* and *dimGrid* to (256, 1) and (104, 1), respectively. There is no need to modify the values, as they are appropriate for all query lengths. After this, the first 26,426 threads perform the filling process, while threads with ID over than 26,425 are not used. Each thread first gets the address number it is in charge of, which is done by simply using its index value. Afterwards, the address number is transferred to a specific word and compared with all query words. Finally, once the comparison score is higher than a threshold value, the position of the word in query sequences is stored and sent back to host memory. Figure 4.7 illustrates the procedure and Figure 4.8 presents the corresponding pseudo code of the procedures.

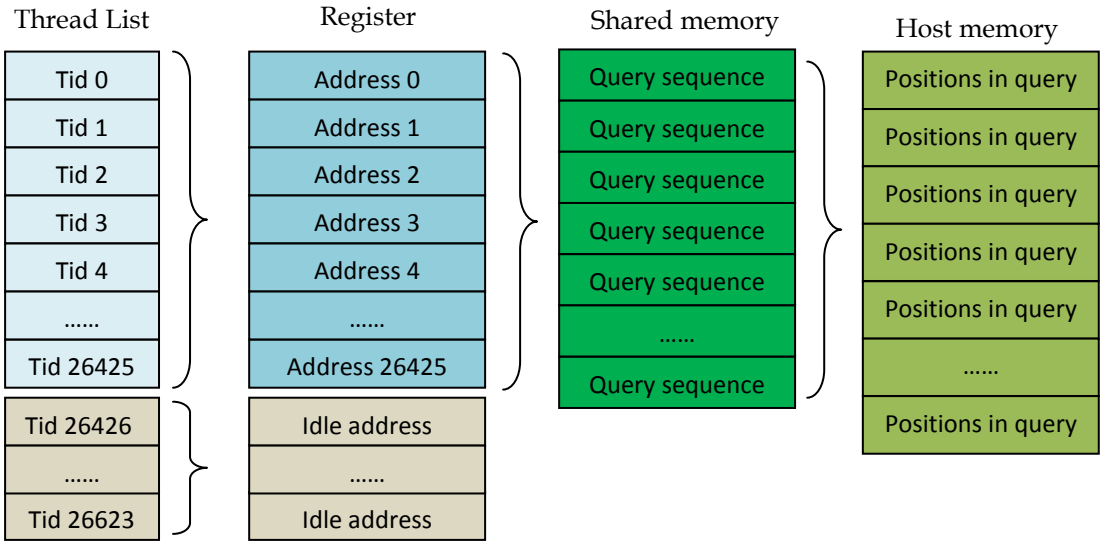


Figure 4.7: The procedure architecture of constructing the lookup table

```
dimBlock(256, 1); dimGrid(104, 1);
tid: thread ID;    adr: address;
__shared__ int query[querySize];          __shared__ int BLOSUM[25][25];

For all threads;
1. tid := dimGrid.x*256 + threadIdx.x;
2. adr := tid;                            7. For sub := 0 to 2 do
/*discompose adr into words*/             8. ch := query[sub + index];
2. word_1 := adr >> 10 ;                  9. word := s + word_(sub + 1);
3. word_2 := ( adr >> 5 ) & 31;           10. s := s + BLOSUM[word][ch];
4. word_3 := adr & 31;                    11. endFor;
/*compare words with query*/              12. If s > threshold, store index;
5. For index := 0 to querySize – 1 do     13. endFor;
6. s = 0;
```

Figure 4.8: The Pseudo code of constructing the lookup table

4.3.3 The Parallelism of the two-hit Method

Subject sequences are read by blocks according their unique ID. Threads within each block search the words from subject sequences using the lookup table as explained section 4.3.2 above. As hit searching is an independent process for each thread in the blocks, there is no need for thread synchronization. Thread iteration is applied when the length of the subject sequence is larger than the number of threads defined in blocks. It is generally a good idea to define a small number of threads in blocks, which reduces the probability of idle state threads i.e. threads with no operation in hand, as threads within the same warp run under the Single Instruction Multi Data (SIMD) model. Figure 4.9 shows a simplified multi-threaded architecture of the second kernel. Each three amino acids from the subject sequence are packed into a word and transformed to a 15-bit offset value. The offset is then used to access the lookup table for hit searching in the query sequence.

The lookup table is constructed and copied from the host memory to the GPU device memory in advance. HIT1 FINDER in Figure 4.9 stands for the process of searching the initial hit in the two-hit method. Within a thread block, it is performed by each thread to traverse all elements on relative diagonals. All operations performed by the threads in dotted lines are implemented in parallel. Once there is a hit occurring, the following HIT2 FINDER process starts to search another hit along the same diagonal of the initial hit within a certain distance. Note that this step is not parallel for all threads as there could be no matched initial hits found in the previous process or the search time for the second hits is not the same for all threads. Therefore, the heuristic layer of seeding influences the efficiency of GPU. The worst situation takes place when there is only one thread entering the extension stage and the rest of threads in the block are kept in the idle state in a thread batch, which leads to 7 idle Stream processors (SPs) in a Stream Multiprocessor (SM) on the GeForce 8800 GTX GPU for instance. This limitation cannot be avoided as BLAST is a heuristic algorithm so that the searching space is determined by the similarity

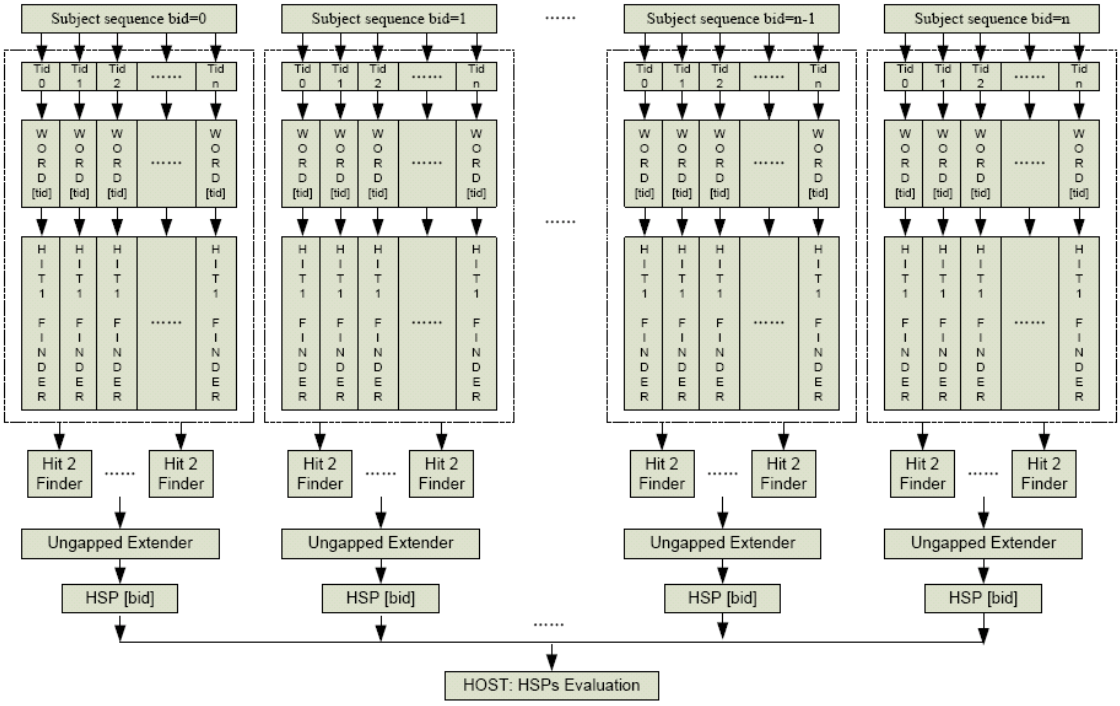


Figure 4.9: Architecture for the Gapped BLAST Algorithm with Two-Hit Method (kernel 2)

between query sequence and subject sequences, which is unpredictable. After the seeding procedure, UNGAPPED EXTENDER processes the sequences with two-hit condition triggered and passes the start and the end points of the high-scoring pairs to the host. The gapped extension part is done by GAPPED EXTENDER in the last kernel after the evaluation of ungapped alignments passed from this one.

The dotted part in Figure 4.10 shows the simplified design of the Two-Hit Finder. For each thread, words extracted from the subject sequence are read into three registers defined as symbol1, symbol2 and symbol3 through three read operations. These operations can be trimmed to one read operation by changing the structure of the subject sequence. Then, the word is transformed to a 15-bit offset value, with the first symbol stored in the most significant 5-bit. The transformation can be applied by operations bit-shift (<<), and bit-or (|).

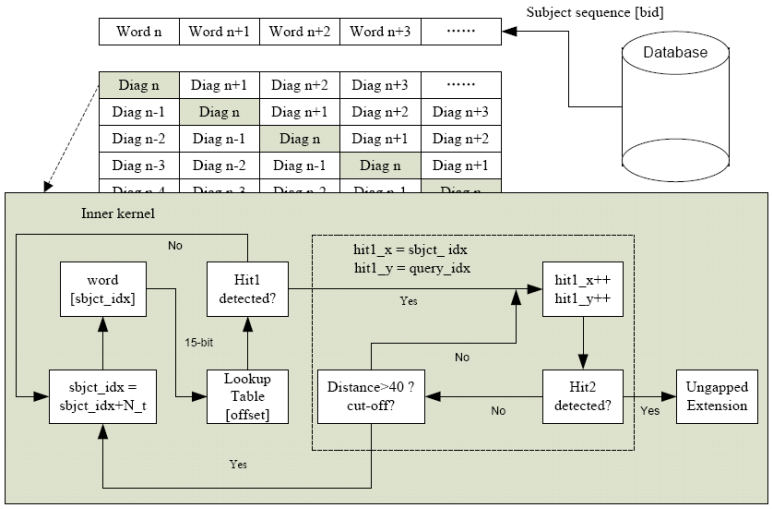


Figure 4.10: Program flow chart of the two-hit finder

The offset value is used to access the lookup table and find where identical or similar words occur. If the flag bit of lookup table is labelled as 1, there is at least one similar word in the query sequence. Its indices are then read and stored in the coordinate registers *hit1_x* and *hit1_y*, respectively. Afterwards, the process searches the second hit along the same diagonal within a certain distance *A* until a hit occurs, or the distance exceeds the size of the scanning window. The ungapped extension score is accumulated simultaneously. Sometimes, the first hit occurs by chance with no other hits found in the same diagonal, which is not uncommon when searching large sequences. Therefore, though a fixed search distance helps searching potential hits, it does not mean that the finally obtained HSPs are high scores. If the Two-Hit condition is satisfied, outward ungapped extension is then trigged and carried in both right and left directions. This extension is terminated when the maximum accumulated score falls a cut-off depth *X1*, below the maximum score so far, or the index of the hits reach the boundary of sequences. Otherwise, threads will search other hits through an offset value based on the index of the last word. Finally, the score of ungapped extension is obtained by adding inward and outward scores, which is passed into the host with the coordinates of the hits

and sequence ID number. Note here that as each pair of sequences is searched by one block, for each thread in the block, the alignment of the thread with the highest score is passed into the stage of gapped extension, while the alignments with lower scores are not [11].

4.3.4 High-Scoring Pairs Evaluation

As illustrated in Figure 4.9, the Two-Hit condition in Gapped BLAST may not occur in some threads. It is possible that many threads in a block return no hits and have been in idle state until the end of the second kernel. In this case, large amount of computing resources are held and they cannot enter the stage of gapped extension if both of the functions are included in just one launch. To avoid this, the second kernel sends the alignments of ungapped extension back to the host first, then those blocks with the Two-Hit condition triggered and whose score are greater than the empirically determined cut-off score S are kept and pushed into the third kernel for gapped extension. The expected number of HSPs with a score at least equal to S is given by Equation 4.8. [12]:

$$\mathrm{E} = \mathrm{k} \times \mathrm{m} \times \mathrm{n} \times \mathrm{e}^{-\lambda \mathrm{S}}; \quad (4.8)$$

where E is the statistical significance threshold for reporting matches against database sequences, with a default value of 10 for large sequences and 1000 for shorter sequences, and mn is the search space [12] which is the product of query sequence length and subject sequence length. If a protein is compared with a whole database rather than a single sequence, n is the total number of residues in the database. The λ and k used in the proposed method are the typical values for ungapped local alignment using BLOSUM62, which are 0.318 and 0.13 [13] respectively.

4.3.5 The Parallelism of Gapped Extender

The Gapped Extender performs the gapped extension on the eligible HSPs after the evaluation. The gapped alignment is implemented by a modified Needleman-Wunsch algorithm. Threads calculate the coordinates of the centre by the edge points. The gapped alignment of the query sequence and related subject sequence is launched in both directions from this point sequentially. As each thread computes a complete HSP, massive threads are harnessed to compute the alignment matrices of different HSPs in parallel. For each alignment matrix, cells are computed column by column. Gapped BLAST requires some modifications to the original Needleman-Wunsch algorithm as the extension is terminated when the maximum value of the current column falls a certain cut-off depth *X2* below the highest value obtained so far. Second, the trace-back procedure starts from the cell with the highest value, rather than bottom rightmost cell.

4.4 Implementation Results and Evaluation

The proposed method was implemented on an NVIDIA GeForce 8800 GTX GPU hosted on a Mac Pro computer with four 2.66GHz Intel Xeon CPU and 8G memory installed. The length of query sequences searched ranges from 64 to 4095, and they were all chosen from UNIPROT database. All query sequences run against the Swiss-Prot protein sequence database [14]. Table 4.1 shows the performance of the proposed method. The cut-off depths for the ungapped extension *X1* and gapped extension *X2* are 11 and 48, respectively. In addition, Table 4.2 shows the performance comparison between the proposed method and the widely used NCBI-BLAST running on CPU. The latter is executed on a 3.4 GHz Pentium4 desktop computer. The figures show a 1.7x to 2.7x speed improvement achieved by the proposed GPU implementation. However, it is worth mentioning that unlike previous GPU implementations of the Smith-Waterman algorithm which resulted in 15x speedups compared to the optimised CPU-based method, the performance benefit for BLAST on GPUs

are relatively smaller, although not negligible. The reason for this is due to the fact that BLAST is a heuristic algorithm, where the operations-to-transfer ratio of BLAST is much smaller than that of the Smith-Waterman algorithm. Indeed,

Table 4.1: Timing performance of the proposed method

sequence ID	*Length*	*E*	*HSPs*	*Time(s)*
1. A4T9V0	64	1000	621	1.2
2. Q2IJ63	128	1000	955	1.7
3. P28484	256	1000	569	2.7
4. Q1JLB7	512	10	405	4.6
5. P08715	1024	10	104	9.7
6. Q8IYD8	2048	10	738	18.1
7. Q06277	4095	10	572	36.7

Table 4.2: Performance comparison between the NCBI BLAST and the proposed method

sequence ID	*Length*	*GPU (s)*	*CPU (s)*	*Speedup*
1. A4T9V0	64	1.2	3.2	2.7
2. Q2IJ63	128	1.7	4.6	2.7
3. P28484	256	2.7	6.4	2.4
4. Q1JLB7	512	4.6	10.4	2.3
5. P08715	1024	9.7	16.5	1.7
6. Q8IYD8	2048	18.1	30	1.7
7. Q06277	4095	36.7	70	1.9

performance benefits can be more readily achieved when the ratio of operations to elements transferred is higher [15]. More precisely, let N denote the length of the query and subject sequences, the size of alignment matrix in the Smith-Waterman algorithm is N^2 and $2N$ elements are transferred into the device from host. Assuming the computation time of one matrix element equals to 1, the operations-to-transfer ratio is thus $N:2$, in which case the larger the matrix, the greater the performance benefits as long as there is large enough bandwidth for data communication and enough threads running simultaneously. However, this parameter is small in BLAST, as BLAST does not explore the entire search space. In addition, BLAST has many conditional

processing which leads to larger threads divergence, which in turn reduces the overall performance on CUDA-compatible GPU. Moreover, the number of hits in each alignment matrix is not fixed and is irregular, so that thread operations cannot be predicted and it is impossible for threads to perform the same operations at the same time especially on large amounts of data. Therefore, the computing time for individual blocks depends on the completion time of the last thread, which decreases the overall performance. This situation would have been even worse if we had merged the second kernel and the third kernel in the proposed method, as data transferred from the host is only used by a small number of threads entering the gapped extension step. Nonetheless, even with this, our proposed method results in a 1.7x-2.7x speed-up compared to the most optimized CPU implementation of Gapped BLAST using CPU and GPU platforms from the same technology (90nm). Given the relatively low cost of GPU platforms and corresponding solution development, this speed-up is not negligible.

4.5 Conclusions

In this chapter, the detailed design and implementation of Gapped BLAST algorithm on GPUs has been presented. Through analyzing the execution of different steps of the algorithm, bottlenecks have been laid out and their effects minimized. The final GPU implementation of our design resulted in a 1.7x-2.7x speed-up compared to the most optimized CPU-based implementation, namely NCBI-BLAST. While this speed-up is smaller compared to the speed-up achieved by GPU implementations of the Smith-Waterman algorithm, it is not negligible especially given the relatively low cost of GPUs. To our knowledge, this is the first GPU-based implementation of the Gapped BLAST algorithm ever reported in the literature.

4.6 References

[1] Smith T.F., Waterman M.S.: Identification of common molecular subsequences. J. Molecular Biology 1981, 147:195–197.

[2] Needleman S.B., Wunsch C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Molecular Biology 1970, 48(3):443–53.

[3] Newell A., Simon H.A.: Computer science as empirical inquiry: symbols and search. Communications Of the ACM 1976. 19:113-126.

[4] Kahneman D., Tversky A. and Slovic P.: Judgment under Uncertainty: Heuristics & Biases. Cambridge University Press ISBN 0-521-28414-7

[5] Clifton A., Ericson II.: Fault Tree Analysis - A History. In the Proceedings of the 17th International Systems Safety Conference,1999.

[6] Altschul S.F., Gish W., Miller W., Myers E.W., Lipman DJ. Basic local alignment search tool. J Mol Biol 1990, 215(3): 403–410.

[7] Pearson W.R., Lipman D.J.: Improved tools for biological sequence comparison.Proc Natl Acad Sci USA 1988 ,85:2444-2448.

[8] Pearson W.R.: Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms.Genomics 1991,11:635-650.

[9] Altschul S.F.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.Nucleic Acids Res 1997, 25:3389-3402.

[10] Korf I., Yandell M. and Bedell J.: An Essential Guide to the Basic Local Alignment Search Tool. Brief Bioinform 2004, 5(1):93-94

[11] Cameron M., Williams H.E. and Cannane A.: A deterministic finite automaton for faster protein hit detection in BLAST. J Comput Biol 2006. 13(4):965-78.

[12] Karlin S. and Altschul S.F.: The Statistic of Sequence Similarity Scores. Proc. Natl. Acad. Sci. USA, 1987.

[13] BLAST. Retrived 4th April, 2012, from http://en.wikipedia.org/wiki/BLAST

[14] Bairoch A., Apweiler R.: The SWISS-PROT protein knowledgebase and its supplement TrEMBL. Nucleic Acid Research, release 56.3, 14-Oct-08.

[15] NVIDIA CUDA C Programming Best Practices Guide Version 2.3, Retrived 4th April, 2012, from http://download.csdn.net/detail/lulyon/2431029

Chapter 5
High Performance Intra-task Parallelization of MSAs on CUDA-compatible GPUs

5.1 Introduction

A Multiple Sequence Alignment (MSA) is used to study the relationships between sets (more than two) of biological sequences. It is the hierarchical extension of pairwise alignment which aims to illustrate the homology or dissimilarity of biological sequences, e.g. protein or DNA sequences. It is particularly useful when studying gene expressions in different species. For instance, this operation is often performed when searching homologies between a newly discovered sequence and a set of known sequences as the functions of unknown species may be inferred from known biological evidence. Traditionally, high quality MSAs were produced by expert biologists, but this work is very tedious and trivial. Automatic MSAs approaches have been proposed and some popular tools have been developed and used widely for phylogenetic analysis. However, MSA is also a prohibitively computationally expensive application as the problem of MSA evaluation can be even NP-hard [1] [2]. Some heuristic approaches have been put forward to make MSAs computation tractable on computers, with some detailed information glossed over however. Progressive alignment [3] is a commonly used approach for MSAs. It restricts the solution to the neighbourhood of only two closest sequences at a time, and carries on the computation between the profile and another sequence until reaching the root of the tree topology produced by a distance matrix. Many popular MSA tools have been developed based on this approach such as T-coffee [4], MUSCLE [5] and ClustalW [6]. The approach can be summarised as a three-stage procedures (see Figure 5.1). Firstly, all sequences are pair aligned to compute nucleotide or amino acid

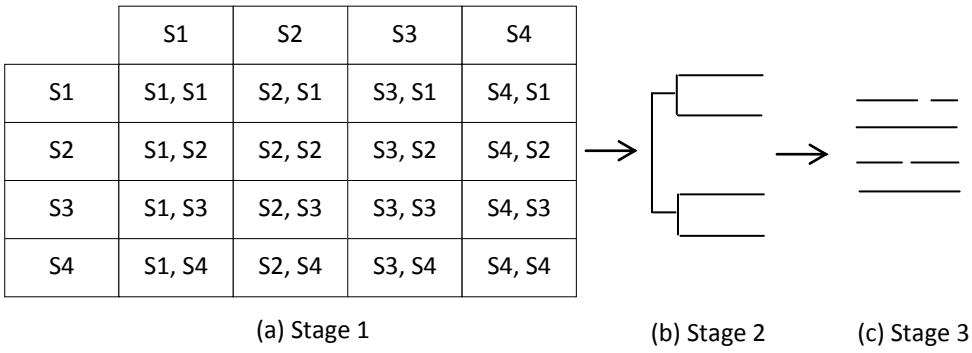


Figure 5.1: (a) Distance Matrix (b) Guide tree (c) Progressive Alignment

differences based on a distance matrix. A guide tree is then generated based on these differences. Afterwards, sequences are aligned to their most similar partners under the same tree nodes using a profile to profile progressive alignment along the branches of the guide tree, until the end of the complete MSA. Although heuristic approaches have been adopted to restrict computing complexity, MSAs still requires hours to process a few hundreds of sequences on state-of-the-art workstations. Given that the size of biological sequence databases is growing at an exponential rate year by year, the demand for more powerful high performance hardware acceleration platforms for MSAs is getting stronger [7]. Graphics Processing Units (GPUs) have been recently proposed as high performance and relatively low cost acceleration platforms for biological sequence analysis [8] [9]. To our knowledge, Liu W.G. et al [10] were the first to present an implementation of MSA on GPU. In [11], Liu Y.C. et al presented an implementation of a complete MSA by two task parallelization approaches (Inter-task and Intra-task) using the Myers-Miller algorithm [12] to deliver optimal alignment in linear memory space on GeForce 280 GTX GPU. The Inter-task parallelization approach achieved considerable speedup factors (up to 40x speed-ups). The Intra-task parallelization approach however achieved lower speedup factors or almost no speedup in some cases (around 1.5x speed-ups). Compared with the Inter-task

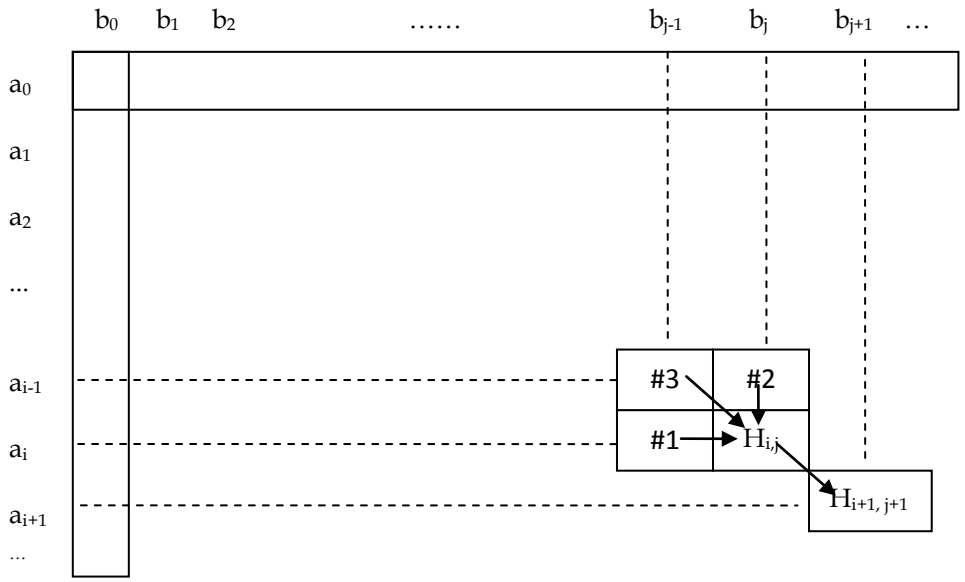
parallelization approach, and as seen in Chapter3, the Intra-task parallelization approach achieves more efficient bandwidth and smaller size of device memory requirements. It is normally used for relatively small sets of sequences where shared cache is commonly exploited for faster data accessing. Inter-task parallelization approach is normally used for relatively large sets of sequences, as device memory access latency can be hidden by many independent threads. The process however needs much more device memory space for each kernel launch. In this chapter, we put forward an improved Intra-task parallelization strategy for MSAs after investigating details about the problem of thread loading imbalance behind the intra-task MSA presented in [11].

The remainder of the chapter firstly introduces background on the Myers-Miller algorithm. Afterwards, the detailed design and implementation of pairwise alignment and our improved strategy in path tracing and progressive alignment are presented. A comparative evaluation of our implementation then follows before conclusions are laid out.

5.2 Background – Essentials of the Myers-Miller algorithm

5.2.1 Computing the alignment cost in linear space

Let $\{a_0\, a_1\, a_2 \ldots a_m\}$ denote a sequence A and let $\{b_0\, b_1\, b_2 \ldots b_n\}$ denote a sequence B. Suppose sequence B is evolved from sequence A, an optimal conversion of A_i to B_j can either result from an insert (#1 in Figure 5.2), a delete (#2 in Figure 5.2) or a replacement of a_i by b_j (#3 in Figure 5.2) illustrated in Figure 5.2. More precisely, #1 indicates sequence B evolved from A with b_j insertion. #2 indicates sequence B evolved from A with a_i deletion. #3 indicates sequence B evolved from A with a_i replacement by b_j. Gotoh [13] presented how to compute the alignment matrix in $O(MN)$ time. Figure 5.3 illustrates the implementation of Gotoh's algorithm in $O(MN)$ space, where symbols C, D and I are equivalent to H, F and E in Equation 3.3 respectively. Figure 5.4 illustrates the cost-only version of Gotoh's algorithm in $O(N)$ memory space with the information of trace path ignored.



$$\begin{array}{lcccccc} & a_0 & a_1 & \ldots\ldots & a_i & - & a_{i+1} \\ \#1 & & & & & & \\ & b_0 & b_1 & \ldots\ldots & b_{i-1} & b_j & b_{j+1} \end{array}$$

$$\begin{array}{lcccccc} & a_0 & a_1 & \ldots\ldots & a_{i-1} & a_i & a_{i+1} \\ \#2 & & & & & & \\ & b_0 & b_1 & \ldots\ldots & b_j & - & b_{j+1} \end{array}$$

$$\begin{array}{lcccccc} & a_0 & a_1 & \ldots\ldots & a_{i-1} & a_i & a_{i+1} \\ \#3 & & & & & & \\ & b_0 & b_1 & \ldots\ldots & b_{j-1} & b_j & b_{j+1} \end{array}$$

Figure 5.2: The three evolution alternatives between two sequences

```
arrays   C[0..M, 0..N], D[0..M, 0..N],
         I[0..M, 0..N]
scalar   t, g, h;
g ← gapOpen; h ← gapExtend;
C(0,0) ← 0;
t ← g;
for j ← 1 to N do
   C(0, j) ← t ← t + h;
   D(0, j) ← t + g;
end for


 t ← g
for i← 1 to M do
   C(i,0) ← t ← t + h;
   I(i,0) ← t + g;
   for j ← 1 to N do
      I(i,j) ← min{I(i,j-1), C(i,j-1) + g} + h;
      D(i,j) ← min{D(i-1, j), C(i-1, j) + g} + h;
      C(i,j) ← min{I(i,j), D(i,j), C(i-1, j-1) + w};
   end for
end for
```

Figure 5.3: Pseudo code of Gotoh's algorithm

```
vectors C[0…N], D[0…N]
scalar e, c, s, t, g, h;
g ← gapOpen; h ← gapExtend;

C(0) ← 0; t ← g;
for j ← 1 to N do
   C( j) ← t ← t + h; D( j) ← t + g;
end for

 t ← g
for i← 1 to M do
   s ← C(0);
   C(0) ← c ← t ← t + h;
   e ← t + g;
   for j ← 1 to N do
      e ← min{e, c + g} + h;
      D(j) ← min{D(j), C(j) + g} + h;
      c ← min{D(j), e, s + w};
      s ← C(j);
      C(j) ← c;
   end for
end for
```

Figure 5.4: The cost-only version of Gotoh's algortihm

5.2.2 Delivering the optimal alignment in linear space

As described in section 5.2.1, the memory space requirement of the Gotoh's algorithm grows quadratically with the size of databases. The implementation is prohibitive when memory space is a limiting factor. Based on this, Hirschberg [14] presented a divide and conquer method to deliver an optimal alignment in linear space. The central idea is to find the midpoint of an optimal conversion by a forward and a reverse cost-only computation on the alignment matrix. The process is continued recursively on both sides of the midpoint until the overall conversion is completed. Suppose two sequences A and B with lengths of M and N respectively. Let i and j be the residue index of A and B. Set i to be $M/2$, so column i properly bisects the alignment matrix. In a forward

phase, the cost-only algorithm is applied to sequence B and a sub-section of sequence A, resulting in vectors C and I satisfying:

$C(j)$ = minimum cost of a conversion of A_i to B_j

$I(j)$ = minimum cost of a conversion of A_i to B_j that ends with a delete

Let $rev(A)$ denote the reverse of A, so sub-section of $rev(A)_{i+1}$ = $\{a_M a_{M-1} \ldots a_{i+1}\}$. Define $rev(B)$ and $rev(B)_{j+1}$ similarly. In a reverse phase, the cost-only algorithm is applied to reverse B and reverse sub-section of A, resulting in vectors R and S satisfying:

$R(N-j)$ = minimum cost of a conversion of $rev(A)_{i+1}$ to $rev(B_j)_{j+1}$

$S(N-j)$ = minimum cost of a conversion of $rev(A)_{i+1}$ to $rev(B)_{j+1}$ that ends with a delete

Let $rev(rev(A))$ denote the reverse of $rev(A)$, so $rev(rev(A)_{i+1})$ = $\{a_{i+1} a_{i+2} \ldots a_M\}$. Let $rev(rev(B))$ denote the reverse of $rev(B)$, so $rev(rev(B))$ = $\{b_1 b_2 \ldots b_N\}$. Vector $S(N-j)$ can also be expressed as the minimum cost of a conversion of $rev(rev(A)_{i+1})$ to $rev(rev(B)j)$ that begins with a delete. This can be illustrated simply as follows:

$a_M \ldots \ldots a_{i+3}\ a_{i+2}\ a_{i+1}$ $\qquad$ $a_{i+1}\ a_{i+2}\ a_{i+3} \ldots \ldots a_M$

$b_N \ldots \ldots b_{j+1}\ \ b_j$ --- $\qquad$ --- $\ b_i\ b_{i+1} \ldots \ldots b_N$

Therefore, for any conversion of A to B, there is a j which belongs to [0, N] such that the conversion is the concatenation of either type 1: a conversion of A_i to B_j and a conversion of $rev(A)_{i+1}$ to $rev(B)_j$ or type 2: a conversion of A_i to B_j that ends with a delete and a conversion of of $rev(rev(A)_{i+1})$ to $rev(rev(B)_j)$ that begins with a delete. Given the vectors above, the midpoint of an optimal conversion

in sequence B can be found by searching the minimum cost of the forward pass and the reverse pass.

$$j = min\begin{cases} C(j) + R(N-j) \\ I(j) + S(N-j) - g \end{cases}; \quad (5.1)$$

Equation 5.1 presents the computation of j. Since discrete short gaps are coalesced into a continuous long gap in type 2 midpoint, the gap penalty value (g) needs to be subtracted. Figure 5.5 illustrates the procedure of searching midpoints, where, area 2 and area 3 are filtered after the first computation.

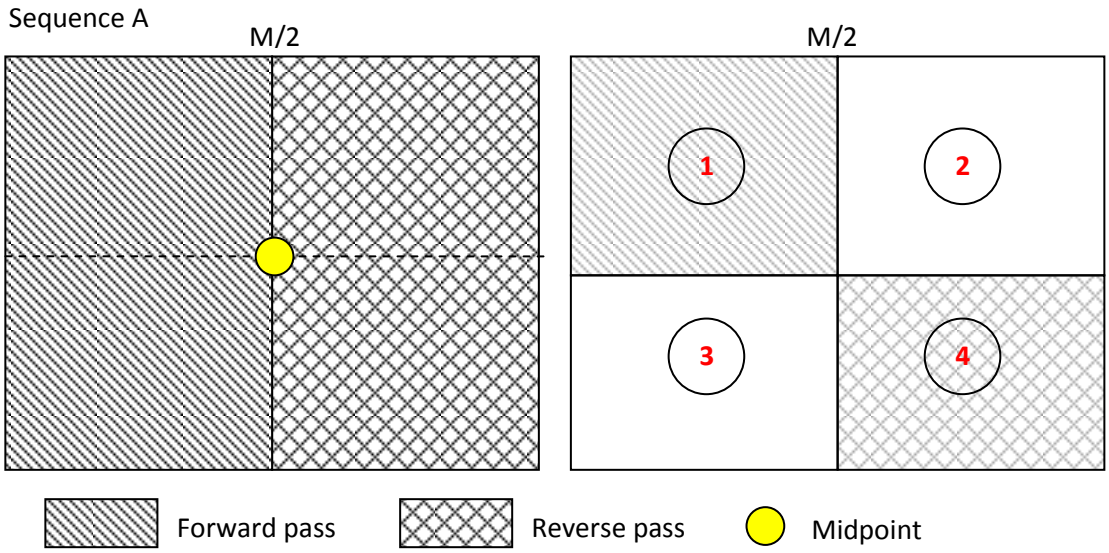


Figure 5.5: Splitting the matrix into sub-matrices by a forward and a reverse pass

5.3 Sequence Distance Computation

5.3.1 Pairwise Alignment

Given two sequences D and Q with lengths L_d and L_q, respectively, their genetic distance can be computed by Equation 5.2:

$$d(D,Q) = 1 - \frac{s\_ch(D,Q)}{min(Ld, Lq)}; \quad (5.2)$$

where $min(L_d, L_q)$ stands for the minimum length between D and Q, $s\_ch(D, Q)$ stands for the number of identical characters in optimal local alignments. $s\_ch(D, Q)$ can be computed by searching the alignment matrix from the position holding the maximum alignment score first zero score is reached. However, since gaps can occur in final optimal alignments, the final alignment score usually cannot be used to stand for the number of the matched characters. Therefore, a different trace back procedure is needed and hence the memory storage requirement changes from $O(L_q)$ to $O(L_q + L_d \times L_q)$ for a single pairwise alignment because of the extra memory space required to record directions of each cell in the alignment matrix in general. The proposed implementation is based on the framework of ClusterW software [6], whereby the optimal local alignment is computed through a forward pass and a reverse pass on the alignment matrix of two different sequences using the Smith-Waterman (SW) algorithm [15]. The actual starting point for the reverse pass procedure is the point holding the maximum score in the forward pass procedure. The number of the matched characters $s\_ch(D, Q)$ between two sequences is counted by a trace back procedure using Myers-Miller algorithm [12], which was developed to get optimal alignments in linear memory space. Sometimes, space, not execution time is the limiting factor when implementing sequence alignment algorithms, which is especially notable on GPU hardware where device memory is limited. Various methods for the SW algorithm implementation on GPU were presented in [16][17]. Figure 5.6 illustrates the pseudo code for the SW algorithm of the proposed implementation. The proposed method is an improved version based on the implementation introduced in chapter 3, where different pairs of sequences is assigned to different warp of threads. For each individual warp, threads cooperate and communicate through shared cache with no thread synchronization procedures. As shown in Figure 5.7, the alignment matrix is divided into parallelograms so that the computation is classified into three stages. Let 2 × 32 shared memory blocks store each sub-section of database sequences and let 32 registers store each sub-section of

```
r_d <— 0;
r_h <— r_e <— 0;
s_h[tid] <— 0;
s_f[tid] <— 0;
for r_i <— 0 to warpSize
{
   r_sbj <— warp_size – 1 – tid + r_i
   r_e <— MAX (r_e – c_gap_ex, r_h – c_gap_oe);
   r_f <— MAX (s_f[tid] – c_gap_ex, s_h[tid] – c_gap_oe);
   r_h <— MAX (r_d + s_M[tid][r_sbj], r_f, r_e, 0);

   r_d <— s_h[tid];
   s_h[tid + 1] <— r_h;
   s_f[tid + 1] <— r_f;
}
```

Figure 5.6: Pseudo code for the implementation of the most inner loop of smith-waterman in Intra-task parallelization, where *tid* denotes the unique id for each thread, *s_M* is the scoring matrix, *r_h*, *r_e* and *r_f* denotes the value from upper-left, left and upper direction respectively.

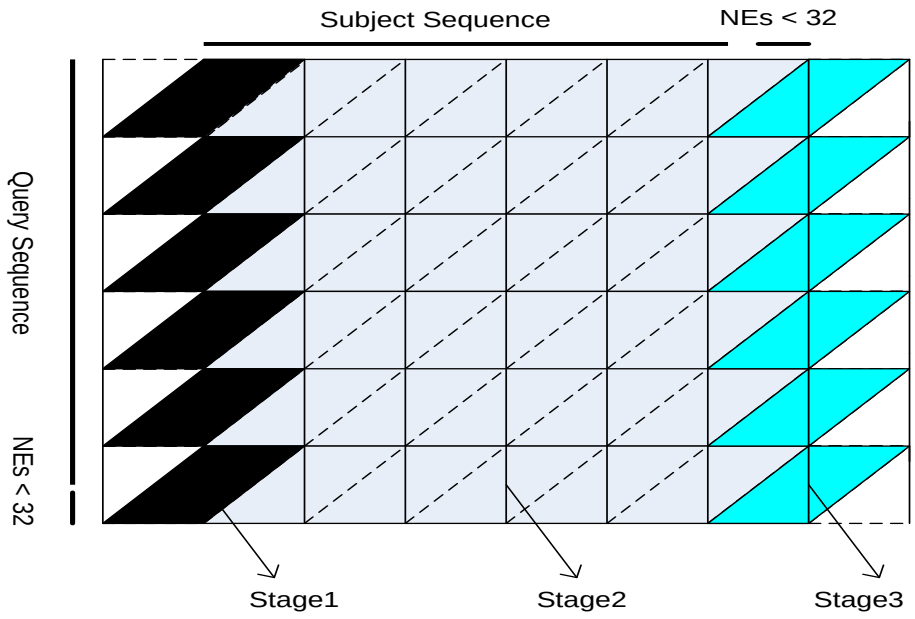


Figure 5.7: The architecture of the computational matrix for each pair of sequences

query sequences. In stage1, the first 32 memory spaces *c[tid]* are initialized as 0, which denotes no input. The following 32 memory spaces *c[tid + 32]* then load

subject characters. The bottom row of the first parallelograms is processed by *tid_31* (Recall that warp size = 32). The range of characters processed is from *c_0* to *c_31*. In stage2, *c[tid + 32]* pass characters into *c[tid]* before they read new characters. In stage3, *c[tid + 32]* pass characters to *c[tid]* before they are set to 0, which denotes dummy characters. Cells in the bottom row of the warp alignment matrix in Figure 5.8 are firstly passed into shared memory by the *32th* thread and written into global memory in two write operations by half-warp of threads afterwards. The pseudo code in Figure 5.6 regarding the computation of the most inner loop is called in each stage with different sub-section of the subject sequence. The intermediate data, *H* and *F*, within each parallelogram are transferred through shared memory. The swap of *H* and *F* for the next sub-section of query sequence is firstly passed into shared memory and then stored in global memory. As shown in Figure 5.8, the computations of diagonal *i* depend on the values of diagonal *i–1* and diagonal *i–2*. Therefore, the overall alignment score in the alignment matrix can be found in linear memory space. After that, the implementation is scaled up to calculate the exact matched characters between the optimal local alignments using the

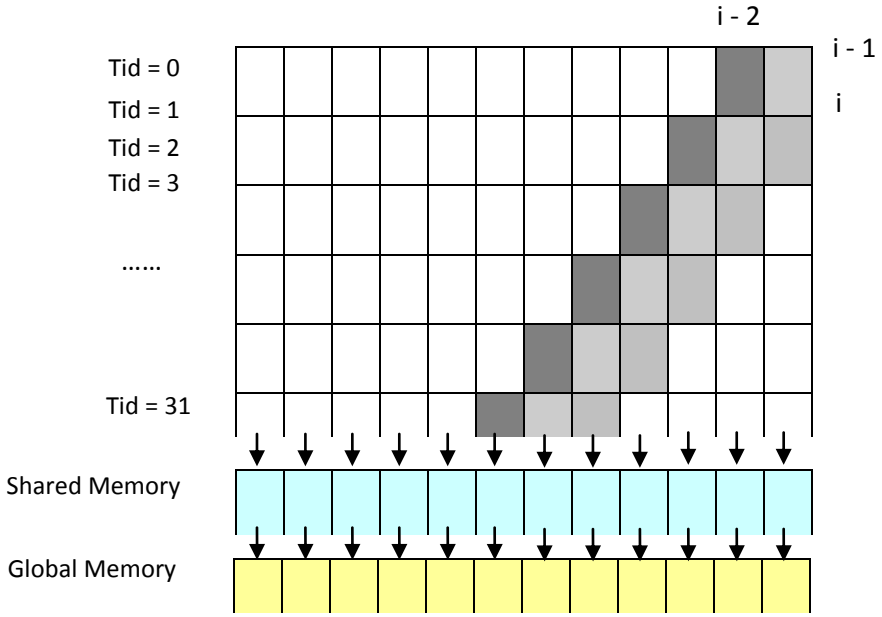


Figure 5.8: Data dependency among cells and memory hierarchies

Myers-Miller algorithm, which was proposed to decrease the memory complexity from $O(L_d \times L_q)$ to linear memory space $O(L_q)$ for optimal alignments.

5.3.2 Searching optimal midpoints

As described above, the local optimal regions of the alignment matrix can be calculated by two pairwise alignment implementations, one forward and one reverse. The actual trace path can be found using the Myers-Miller algorithm. As explained above, the central idea of the algorithm is to search recursively for optimal alignment residues, which are the midpoints of an optimal conversion using a forward and a reverse cost-only pairwise alignment. However, recursive function calls are not supported on CUDA GPUs. Liu Y.C. et al. [11] presented a stack-based iterative method to implement the Myers-Miller algorithm on an NVIDIA 280GTX GPU card. Prior to that, Liu W.G. et al. [10] presented another approach to MSA implementation on an NVIDIA 7800GTX card. Since the proposed technique draws from the experience of both aforementioned works, a brief description of these prior works is firstly given before reporting the details of our own method. In [10], Liu W. G. et al. presented the first MSA GPU implementation, to our knowledge. Liu W. G. et al. however did not use the Myers-Miller algorithm, hence requiring extra memory resources. Moreover, their implementation targeted an NVIDIA 7800 GTX card, which is one generation behind the GPU card we targeted in this work. Based on Equation 3.3, Equation 5.3 below gives the number of matched characters between two sequences:

$$N(i,j) = \begin{cases} 0 & if\ H(i,j) = 0 \\ N(i-1,j-1) + m(i,j) & if\ H(i,j) = H(i-1,j-1) + Wi,j \\ N(i,j-1) & if\ H(i,j) = E(i,j) \\ N(i-1,j) & if\ H(i,j) = F(i,j) \end{cases} \tag{5.3}$$

Where $m(i, j)$ is equal to 1 if $d_i = q_j$ and 0 otherwise. From Equation 5.3, the maximum $N(i, j)$ stands for the number of matched characters in $O(L_q * L_d)$ memory space. Since the memory consumption for this approach grows quadratically with the length of sequences, the number of parallelizable threads is limited in this method. Therefore, this approach can be performed, but definitely infeasible to achieve high performance acceleration results. Performance significantly degrades when aligning datasets having long length of sequences.

In [11], Liu Y. C. et al. presented two task allocation strategies for parallelizing pairwise alignments. Both of them adopted the Myers-Miller algorithm to conquer the problem in linear memory space. These are Inter-task parallelization strategy and Intra-task parallelization strategy respectively. The former harnesses single thread to process a specific problem in an application, while the later harnesses a block of threads to divide a specific problem into sub problems with each thread processing relevant sub problems. In [11], Intra-task parallelization resulted in lower speedup factors or almost no speedup in some cases compared to Inter-task parallelization. The major problem of Intra-task parallelization approach for computing optimal midpoints in this implementation is caused by the insufficient various numbers of cells for each recursion. For instance, given two sequences D and Q, composed of identical characters and having the same length L (L = 40), the computation order of midpoints in both D and Q and the number of characters involved in each recursive operation are shown in Figure 5.9 and Figure 5.10 respectively. The maximum size and the minimum size of the matrix computed in the forward pass and reverse pass are 1600 and 4 respectively. Therefore, the number of characters involved varies considerably and given a fixed number of threads, loss of performance is inevitable because of thread load imbalance in the search of optimal midpoints. The computation leads to idle operations of threads when the number of threads is bigger than the characters in each sub section of D or Q. The actual thread usage ratio

degrades to 1 / N when computing a sub-section with only one character, where N is the number of threads allocated to process each sub matrix.

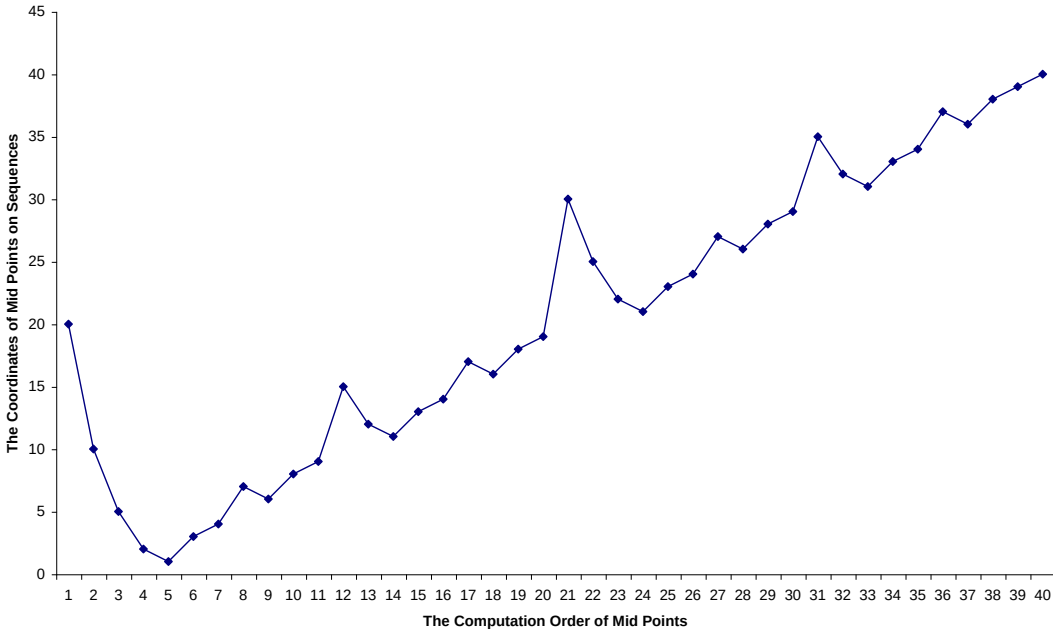


Figure 5.9: The coordinates of mid points on sequences

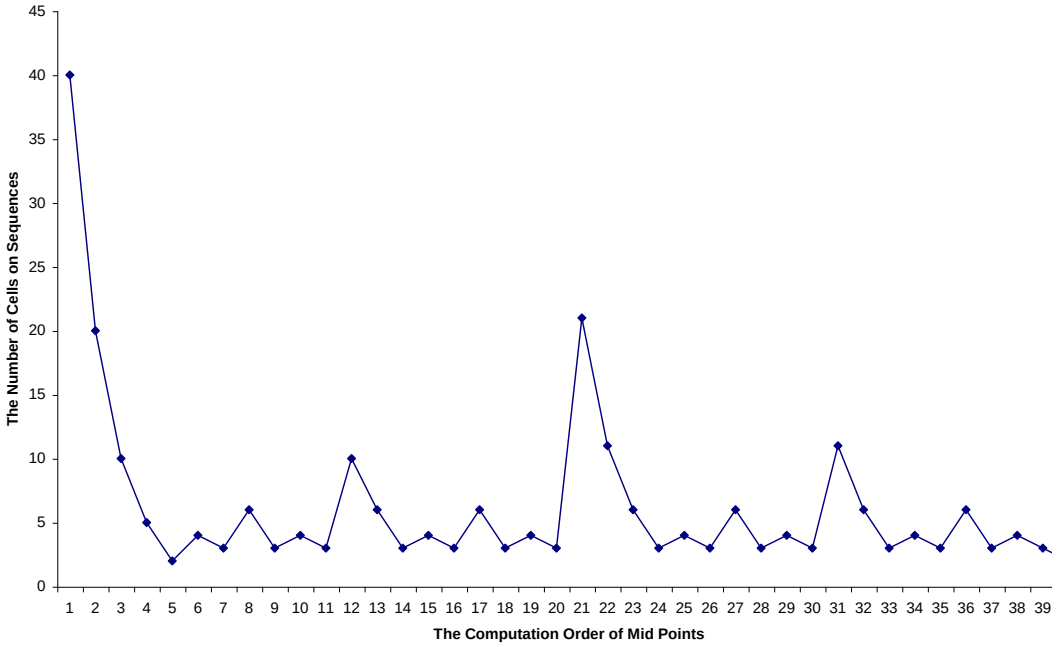


Figure 5.10: The number of characters needs to be calculated on one sequence.

In this chapter, an improved Intra-task parallelization strategy is proposed. Above all, one of the sequences in the sequence pair is chosen and recursively divided into segments of equal lengths. This operation stops once the length of the final segment is shorter than a threshold value *T*. As such, the number of the potential blocks separated from the entire matrix can be predicted. The optimal midpoints in another sequence can be calculated one by one using Myers-Miller algorithm. Then, cells in each sub-matrix are computed with their alignment directions stored in shared memory so that the trace path can be accessed later. The strategy uses the Intra-task parallelization approach and Myers-Miller algorithm. Compared with the implementation described in [11], it does not compute all midpoints as a profile for the requirement of trace path can be obtained by concatenating several points on sequences. Figure 5.11 shows the approach presented in [11]. The red nodes stand for the optimal mid points, connecting all mid points produces the final optimal alignment. With all midpoints computed, intra-task parallelization leads to many thread load imbalance in MSAs. Figure 5.12 shows our proposed approach for CUDA GPU implementations. Firstly, a subject sequence in a pair alignment is divided by the number of threads until the length of the final segment is smaller enough so that its matrix alignment can be stored in shared memory. The number of

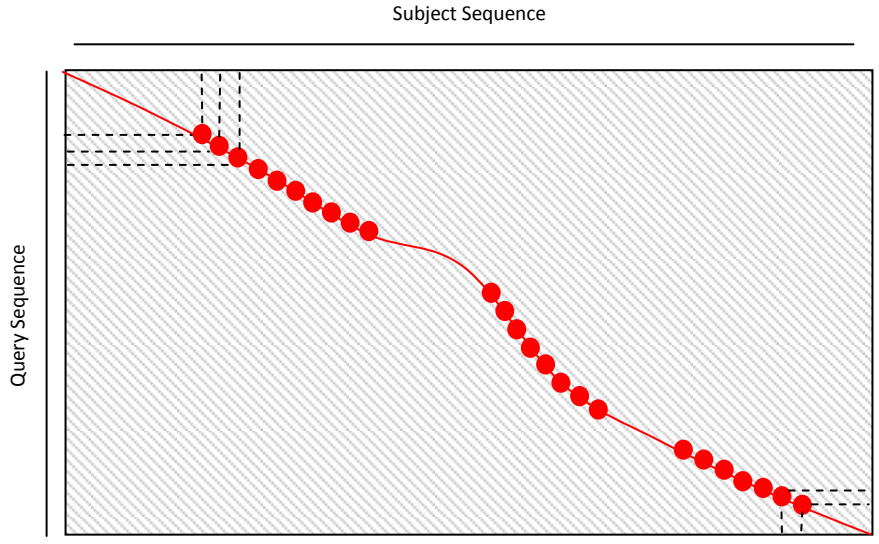


Figure 5.11: Trace path of the alignment matrix

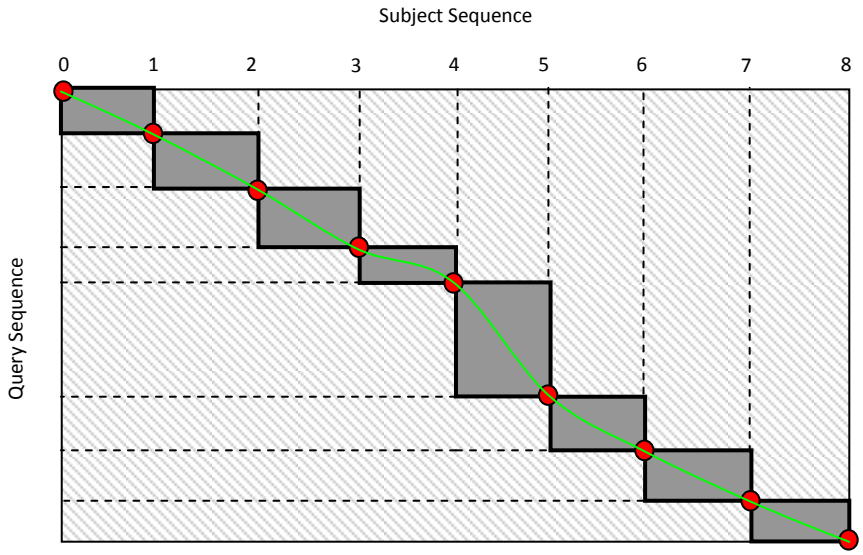


Figure 5.12: Trace belt of the alignment matrix

midpoints can be deduced from the number of divisions performed. Secondly, midpoints on the query sequence corresponding to those on subject sequences in the same pair can be found through forward phase computations and reverse phase computations. A trace belt (see shaded area in Figure 5.12) is then obtained by linking all corresponding midpoints on the sequences. Until now, the actual trace path still has not been found, but it is obvious that the trace path should be within the trace belt. In some cases, the interval value between two neighbouring midpoints in query sequences is bigger than the number of threads, which is possible if there are enough gaps in subject sequences. Therefore, the query segment needs to be divided until it is small enough for shared memory data storage. Blocks composing the trace belt are then computed one by one with directions stored in shared memory. The overall trace path is the concatenation of all small trace paths in sub blocks.

Compared with the approach described in [10], the improved intra-task approach allocates fixed size of shared memory to store trace path for each sub matrix. Shared memory consumption only depends on the number of threads assigned to that pair, instead of the dimension of the alignment matrix

involved by sequences. Suppose 8 threads are assigned to process one sub trace belt, the necessary memory storage is 8×8 = 64 bytes regardless of the sequence lengths. Figure 5.13 presents the pseudo code for computing the trace belt. Firstly, the number of sub blocks is calculated using the length of subject sequence. This is achieved by recursively dividing the subject sequence in two until the length of resulting sub-sequence is equal to or smaller than the width of matrix in shared memory. *r_No = 1* means that the final number of sub-blocks is equal to 1 if the length of query sequence is small enough. Otherwise, the size of alignment matrix outstrips the storage of shared cache. In this case, the whole matrix is loaded and passed by the forward and reverse computation to find the optimal mid-point on query sequence after the

```
1. sbj_num <— divider(sbj_size);
2. distance <— mid <— sbj_num / 2;
3. sbj[0] <— 1;
4. sbj[sbj_num] <— sbj_size;
5. que[0] <— 1;
6. que[sbj_num] <— que_size;
7. stack_idx <— 0;

8. while(sbj_num > 1)
   {

9.    sbj_h <— sbj[mid - distance];
10.  sbj_t <— sbj[mid + distance];
11.  que_h <— que[mid - distance];
12.  que_t <— que[mid + distance];
13.  sbj[mid] <— (sbj_t – sbj_h) / 2;

14.  forwardPass(sbj_h, sbj[mid]);
15.  reversePass(sbj[mid],   sbj_t);

16.  max_score <— hh[0] + rr[0];
17.  que[mid] <— que_h;

18. for idx <— que_h to que_t do {

19. t1 <— C[idx] + R[idx];
20. t2 <— I[idx] + S[idx] + gap_open;
21. t =MAX(t1, t2);
22. if(max_score < t)
      {

23.     max_score <— t;
24.     que[mid] <— idx;
      }
   }
   25. Update mid and distance;
}
```

Figure 5.13: Pseudo code of computing optimal mid points in our proposed method

determination of the midpoints on subject sequence. The latter is defined as half the size of the subject sequence in the first computation. *C*, *I*, *R* and *S* are

vectors allocated in CUDA global memory, where, vector *C* and vector *R* are used to store the *H* value in the column determined by the midpoint on the subject sequence. Vector *I* and vector *S* are used to store the *E* value in that column. Rows having the maximum score are the optimal points in the query sequence. There is no need to store *F* values as the cell with the maximum score which had been determined in that column could only have come from its left neighbour or from its left-upper neighbour. Once the optimal points on query sequence and subject sequence are found, the preliminary trace profile is then obtained. The temporary midpoints on the subject sequence need to be pushed into a stack defined in shared memory for the purpose of making computations in the right order.

5.3.3 Optimal alignment trace back

Up to this point, the full trace path has not found yet as not all midpoints have been computed. Figure 5.14 presents the pseudo code of alignment trace back for each sub-block. Though preliminary trace profile composed by sub blocks has been found, sub-blocks still cannot be put into shared memory if their heights are larger than the number of threads, which occurs when the length of the sub query sequence is longer than the corresponding length of the sub subject sequence and optimal alignments need to insert gaps in subject sequences. This is not common in homogenous sequence databases, especially in local alignments. However, for databases composed of heterogeneous sequences, there are usually gaps in a pair of optimal alignment. Here, it is necessary to check the sub-length of query sequence for the purpose of doing the final alignment by executing the pseudo code in Figure 5.13. Here, we divide the query sequence rather than the subject sequences. Afterwards, each sub-block is passed through the pseudo code in Figure 5.14 with the directions of aligned cells stored in shared memory. In the proposed approach, constants 1, 2 and 3 are used to denote the left, upper and upper-left directions respectively. The starting point is initialized as 0. Finally, the optimal

alignment can be found by tracing from the cell in the bottom-right corner to the cell in the upper-left corner in the shared memory. The shared memory is then used iteratively for the next sub-matrix.

```
1. __shared__ char trace[warp_size][warp_size];
2. trace[0][0] <— 0; trace[threadId][0] <— 0;
3. trace[0][threadId] <— 0; match_num <— 0;

4.  if(h < d + w)
    {
5.    h <— d + w;
6.    trace[threadId + 1][idx] = trace[threadId][idx - 1] + M(w);
    }
7.  if(h < e)
    {
8.    h <— e;
9.    trace[threadId + 1][idx] = trace[threadId + 1][idx - 1];
    }
10. if(h < f)
    {
11.   h <— f;
12.   trace[threadId + 1][idx] = trace[threadId][idx];
    }

13.   match_num <— trace[que_size][sbj_size];
```

Figure 5.14: Pseudo code of computing matched characters of sequences in a pair. Vectors recording trace path are needed in the stage of progressive alignment.

5.3.4 Guide tree

The Neighbor-Joining (NJ) method [18] is used to construct an unrooted tree. Exhaustive distances between nodes are traversed from the distance matrix to identify two closest nodes at each step. For a set of sequences *N*, the number of distances between each pair is *N * (N - 1) / 2*. The traversal finds the smallest distance value and combines two closest nodes into a new node until *N* is equal to 1. An unrooted tree is produced by the NJ method. Unrooted trees can be rooted through the approach presented in [19]. Since the percentage of time consumed on the construction and rooting of guided trees is relatively small

compared with stage 1 and stage 3 of MSA (see Figure 5.1), the acceleration of this stage does not have the same effect on the overall speedup. For the sake of simplicity, we thus implemented this stage on the host. Figure 5.15 illustrates the production of an unrooted tree from a distance matrix.

5.3.5 Parallelization of Progressive Alignment

The stage of progressive alignment performs profile-profile alignment on the guided tree starting from leaf nodes up to the root node. Leaf nodes stand for the original sequences which are the initial descendent nodes composing internal nodes. The latter are produced by two aligned nodes. For each internal node, there are three combination alternatives on their descendents, namely 1) two sequences, 2) two nodes, and 3) one node with one sequence. Internal nodes can be aligned only when their left sub node and right sub node have been performed. Therefore, the computation order of nodes can be parallelized using parallel processing batches. Nodes with their left sub node and right sub node performed can be computed in the same batch. In the proposed method, the progressive alignment of the guided tree can be performed by the following steps: Firstly, a dependency structure corresponding to each node of the guided tree is created and used to store the attributes of each node. For each internal node, the structure contains its left descendent, right descendent and an aligned flag bit which denotes whether the node is in the waiting queue or not. For each leaf node, its left children and right children are set to 0 as it has no descendents. The flag bit is set to 1 automatically. The first launch is always performed on the internal nodes where both descendents are sequences. Afterwards, the pairwise alignments of all profile pairs are performed on GPU in parallel using the 'trace belt' approach which was presented in a previous section. The pseudo code in Figure 5.14 presented the trace back procedure on each sub-trace shown in Figure 5.12. For the progressive alignment, there are some differences with the pseudo code presented in Figure 5.14, as not only optimal alignment scores but also optimal alignments need to be passed to the

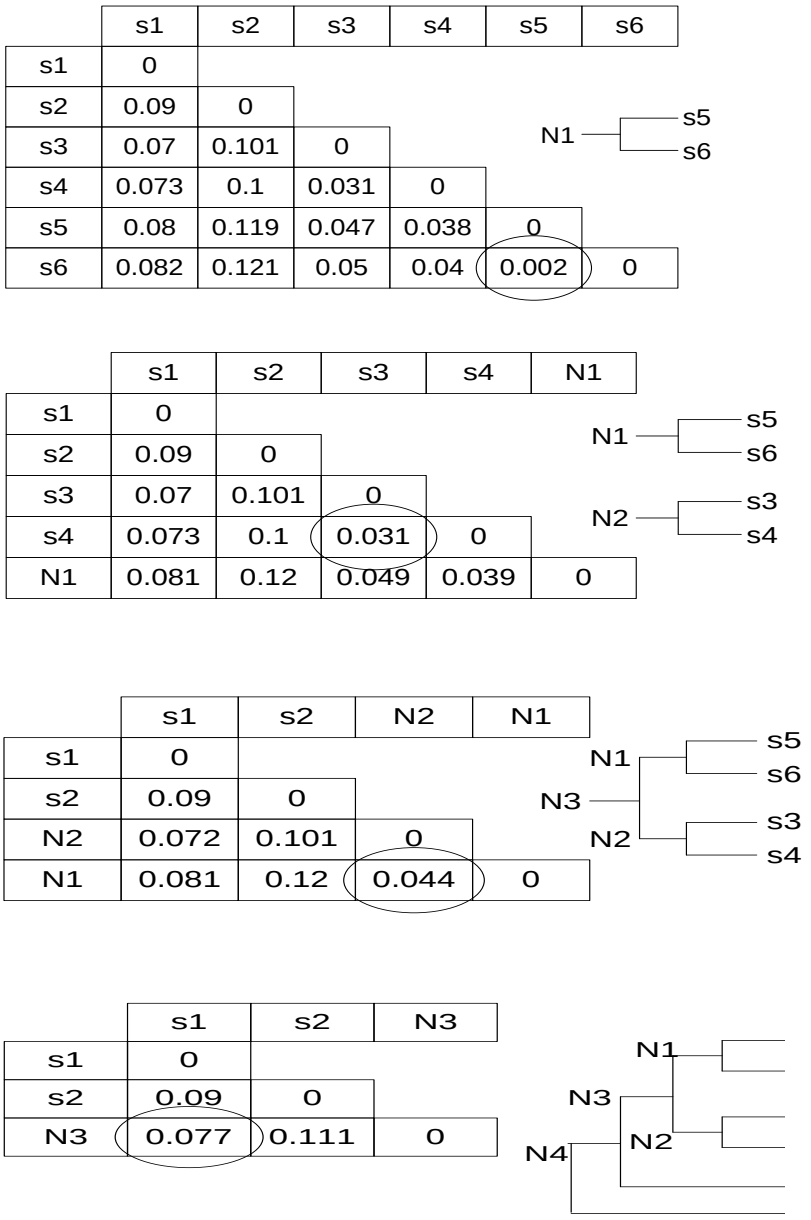


Figure 5.15: Construction of the guided tree by the Neighbour Joining Method, after joining two species into a new node, the distance from every other node needs to be computed.

host in this instance. Hence, for each pair of profiles, the number of gaps between two alignments is accumulated exactly on specific positions and stored in two gap list vectors in global memory. Based on these lists, optimal alignments corresponding to nodes in pairs can be constructed on the host. The aligned descendents of nodes are packed into new profiles. Their aligned flags are then set to 1 and are added into a ready queue for the next launch. These operations are performed iteratively until all internal nodes are aligned. As shown in Figure 5.16, nodes with the same colour can be processed in parallel by different blocks and threads.

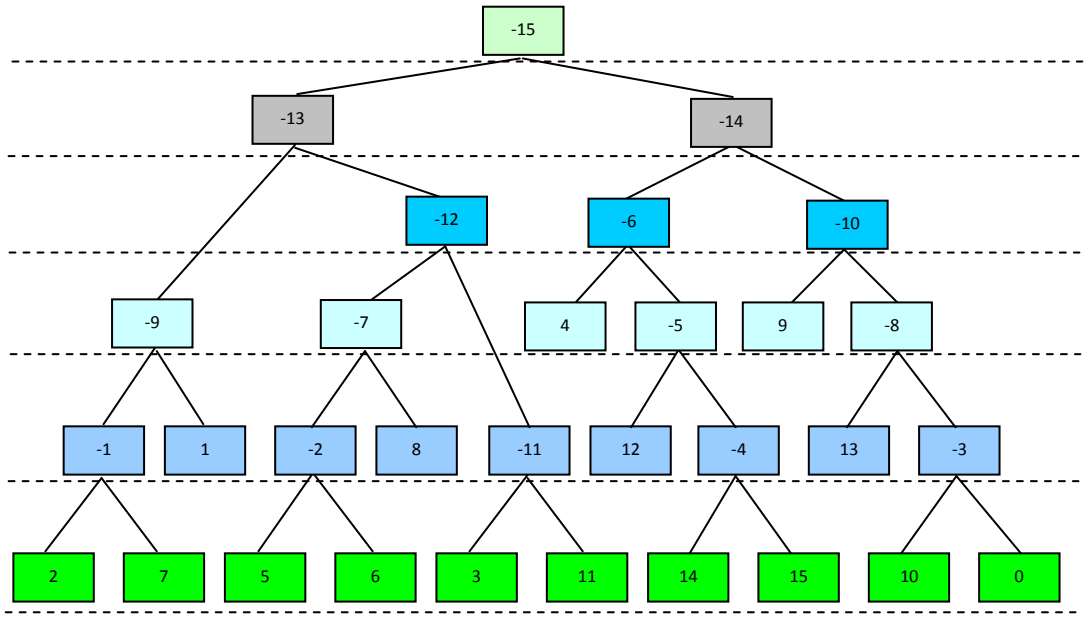


Figure 5.16: Example of a tree topology, nodes with the same colour are aligned in parallel.

5.4 Performance Evaluation

The proposed method was implemented on an NVIDIA GeForce 8800 GTX GPU installed on a Mac Pro machine with four 2.66GHz Intel Xeon CPU and 8G memory installed processors. The evaluations are separated into three test groups (long, medium, short sequences of database sizes) as shown below.

- G1: long DNA sequences.
- G1.1: small size database, 50 sequences of average length 865.
- G1.2: medium size database, 400 sequences of average length 868.
- G1.3: large size database, 1000 sequences of average length 867.

- G2: medium protein sequences.
- G2.1: small size database, 100 sequences of average length 433
- G2.2: medium size database, 500 sequences of average length 438
- G2.3: large size database, 2000 sequences of average length 438

- G3: short protein sequences.
- G3.1: small size database, 100 sequences of average length 89
- G3.2: medium size database, 500 sequences of average length 89
- G3.3: large size database, 2000 sequences of average length 89

Biological sequences from [20] are evaluated. The version of ClustalW software used for comparison is ClustalW 2.0.12. Figure 5.17 illustrates performance comparison between the ClustalW and the proposed method for stage 1. This shows that the proposed method for stage 1 achieves considerable speedup ranging from 18.3x to 21.3x for G1, 12.3x to 13x for G2 and 7x to 8.6x for G3. Hence, the speedup is proportional to the number of sequences and sequence length, which is similar to the trend presented in [11] when using the Inter-task parallelization. Therefore, considerable speedup can be achieved when there are enough tasks for both Intra-task parallelization and Inter-task parallelization approaches A direct and fair comparison between our implementation and that presented in [11] is however not possible since we used different GPU platforms. In general, the Intra-task approach is chosen for a relatively smaller number of tasks, whereas the Inter-task approach performs better for larger number of tasks, as explained in Chapter 3. Figure 5.18

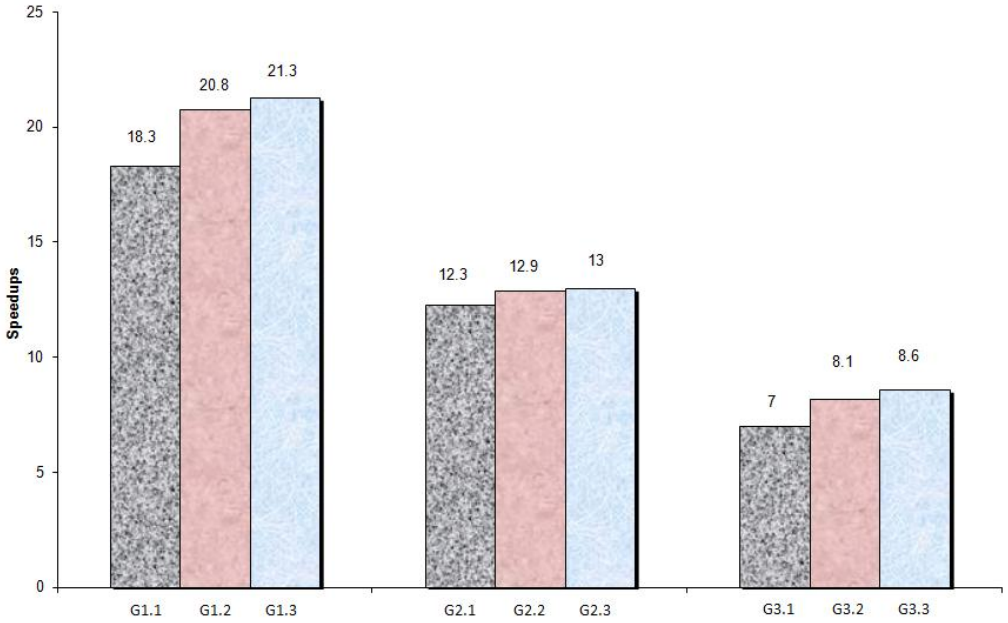


Figure 5.17: Performance comparison between ClustalW and the proposed approach for stage1

illustrates performance comparison of stage 3. This shows that the speedups for stage 3 are lower, ranging from 2x to 4.4x for G1, 1.8x to 3.5x for G2 and 1.5x to 2.8x for G3, as the number of nodes aligned in parallel in CUDA depends heavily on the tree topologies. Higher performance can be gained once the GPU hardware is fulfilled with enough nodes. In addition, the construction of profiles needs to be done sequentially on the host, with the additional data transfer overhead between host and device.

Table 5.1 presents performance comparisons between the Intra-task implementations presented in [11] and our improved Intra-task parallelization approach for G1.1, G1.2 and G1.3. This shows that the performance of the improved approach outperforms the same parallelization approach presented in [11] even on an older type of GPU. We do not claim that the proposed Intra-task parallelization approach outperforms the Inter-task parallelization approach presented in [11] as the implementations are carried on different hardware platforms. Nonetheless, the experimental results give similar

arguments to the discussion of Intra-task parallelization and Inter-task parallelization approaches in section 5.1.

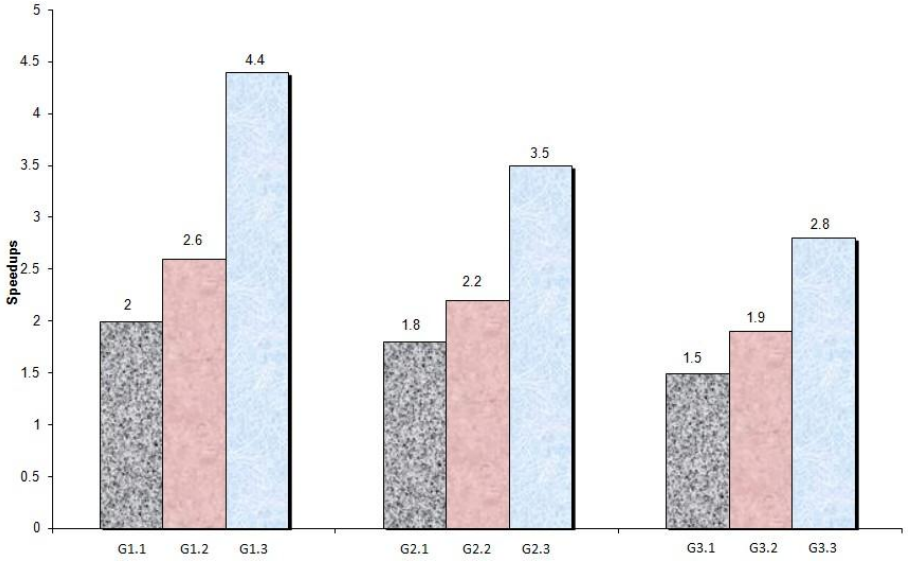


Figure 5.18: Performance comparison between ClustalW and the proposed method for stage 3

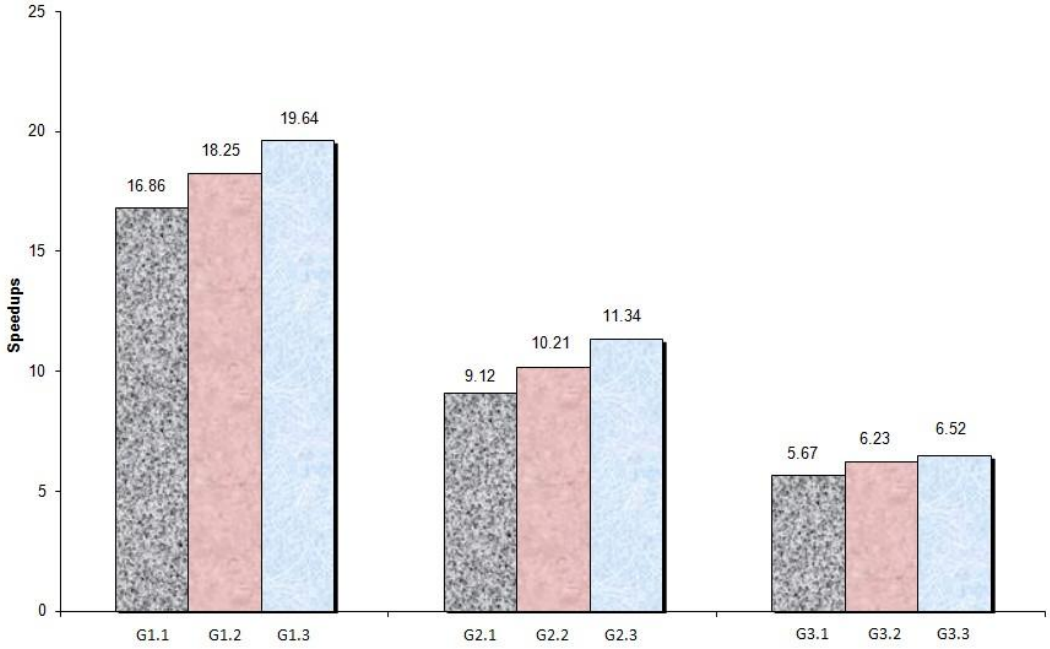


Figure 5.19: The overall speedups for stage 1 and stage 3 between ClustalW and the proposed approach

Table 5.1: Performance comparison between MSA-CUDA [11] and the proposed approach

The number of sequence	Speedup[1]	Speedup[2]	Speedup[3]
50	18.3	N/A	N/A
100	18.1	N/A	N/A
200	18.1	N/A	N/A
300	18.7	N/A	N/A
400	20.8	10.43	45.16
1000	21.0	10.44	47.13

Speedup[1]: Performance comparison between the proposed method and ClustalW
Speedup[2]: Performance comparison between Intra-task approach presented in [11] and ClustalW
Speedup[3]: Performance comparison between Inter-task approach presented in [11] and ClustalW

5.5 Conclusions

In this chapter, we presented an improved strategy to exploit the Intra-task parallelization approach in the implementation of multiple sequence alignments on CUDA-compatible GPUs. This approach harnesses multiple threads to conquer the problem in a divide and conquer approach. The proposed strategy improves the efficient operations of threads and decreases the redundant computations reported in previous GPU implementation of MSAs. Central to this is a divide and conquer approach to alignment matrix calculation in which the trace path of sub-matrix is stored in shared memory resources on GPU hardware. Through the parallelization of two stages of the ClustalW MSAs, the proposed GPU implementation outperforms an optimised CPU-only implementation by factors ranging from 6.2x to 19.6x as shown in Figure 5.19. Moreover, the best reported speedup in the proposed approach doubled the performance over a previous implementation of MSAs even on an older type of GPU.

5.6 References

[1] Wang L. and Jiang T.: On the complexity of multiple sequence alignment. J Comput Biol, 1994, 1(4):337-348.

[2] Elias I.: Settling the intractability of multiple alignment. J Comput Biol, 2006, 13(7):1323-1339.

[3] Feng D. and Doolittle R.: Progressive sequence alignment as a prerequisite to a correct phylogenetic Trees. J. Molecular Evolution 1987, 25:351–360.

[4] Notredame C., Higgins D.G. and Heringa J.: T-Coffee: a novel method for fast and accurate multiple sequence alignment. J. Mol. Biol 2000, 302:205-217.

[5] Edgar R.C.: MUSCLE: a multiple sequence alignment method with reduced time and space complexity. BMC Bioinformatics 2004, 5:113

[6] Thompson J.D., Higgins D.G. and Gibson T.J.: CLUSTALW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res 1994, 22:4673–4680.

[7] Durbin R., Eddy S.R., Krogh A., Mitchison G.: Biological Sequnce Analysis: Probabilistic Models for Proteins and Nucleic Acids. Cambridge University Press, Cambridge University UK, 1998.

[8] Charalambous M., Transcoso P. and Stamatakis A.: Intial experiences porting a bioinformatics application to a graphics processor. In Processings of 10th Panhellenic Conference on Informatics 2005, pp.415-425.

[9] Varre J.S., Schmidt B., Janot S. and Giraud M. : Many-core high-performance computing in bioinformatics. In Advances in Genomic Sequence Analysis and Pattern Discovery 2011, pp.179-207.

[10] Liu W.G., Schmidt B. and Voss G.: GPU – ClustalW: Using Graphics Hardware to Accelerate Multiple Sequence Alignment. HIPC 2006, pp.363-374.

[11] Liu Y.C., Schmidt B. and Maskell D.L.: MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. In the processing of 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2009, pp.121-128.

[12] Myers E.W. and Miller W.: Optimal alignments in linear space. Comput Appl Biosci, 1988, 4:11-17.

[13] Gotoh O.: An improved algorithm for matching biological sequences, J. Molec. Biol, 1982, 162:705-708.

[14] Hirschberg D.S.: A linear space algorithm for computing longest common subsequences, Commun. Assoc. Comput, 1975, 18:341-343.

[15] Smith T.F., Waterman M.S.: Identification of common molecular subsequences. J Molecular Biology, 1981, 147:195–197.

[16] Liu Y.C., Maskell D.L. and Schmidt B.: CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Research Notes, 2009, 2(1):73

[17] Manavski S.A., Valle G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 2008.9(Suppl2):s10

[18] Saitou M. and Nei N.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol, 1987, 4:406-425.

[19] Thompson J.D., Higgins D. and Gibson T.J.: Improved sensitivity of profile searches through the use of sequence weights and gap excision. Comput. Appl. Biosci, 1994, 10:19-29.

[20] Database, 12th April 2012 retrieved from http://www.ncbi.nlm.nih.gov/

Chapter 6
High Performance Phylogenetic Analysis on CUDA-compatible GPUs

6.1 Introduction

In biology, phylogenetic trees are used to represent the evolutionary relationships among groups of organisms or among a family of related nucleic acid or protein sequences based upon their similarities and differences. They are helpful in inferring the history of organism lineages as they evolve over time. Since phylogenetic relationships among species also can help determine which ones might have similar functions, it is used widely in medical research for predicting potential hazards from species with rapidly changing structures, for example indentify the variant of HIV and SARS virus. Phylogenetic trees are normally branching diagrams, where, leaves in trees represent species, and species are joined together to compose internal nodes. Internal nodes in trees represent the inferred most recent common ancestors of their descendents. There are some computational phylogenetic methods developed to construct phylogenetic trees, such as Neighbor-Joining method (NJ) [1] or UPGMA [2]. The idea behind the NJ method is to create a matrix which includes all pairs of biological sequences with size $N \times (N-1)/2$, where, N stands for the number of sequences. Afterwards, it calculates the ratio of identical characters and the minimum sequence length between each pair of sequences. The phylogenetic tree is finally constructed based on the distance. However, *NJ* method only gives one possible tree and strongly depends on the evolutionary model used. More advanced methods to infer phylogeny have been put forward, including Maximum Likelihood (ML) [3] and Maximum Parsimony (MP) [4]. Their basic idea is to search exhaustively for the tree with maximum likelihood or maximum parsimony over all possible trees. Since the number of possible trees grows in a factorial way, the identification of the optimal tree is

computationally prohibitive. Therefore, heuristic and optimization approaches are developed to conquer the problem. This chapter presents the detailed GPU-based multi-threaded design and implementation of the ML method for phylogenetic analysis on a set of aligned amino acid sequences, which is based on the framework of the most widely used software named MrBayes.

The remainder of the chapter first present essential background of phylogenetic analysis and ML algorithm. Afterwards, the Bayes theorem and Markov Chain Monte Carlo method are presented. The detailed design and implementation of MCMC-based phylogenetic analysis on CUDA-compatible GPU is then presented. A comparative evaluation between the proposed GPU-based implementation and GPP-based MrBayes software [5] follows before conclusions are laid out.

6.2 Background

6.2.1 Phylogenetic Analysis

Phylogenetic analysis aims to infer the evolutionary order of a set of species or genes. It estimates the relationship by comparing homologous sites between species on different branches so that evolutionary scores can be assigned to given phylogenetic trees. Since biological sequences normally have some dissimilarities, sequences under investigation need to be multiply-aligned so that homologous sites can be discovered to form columns in the alignment. These alignments are then used to construct phylogenetic trees. Phylogenetic tree is a diagram which depicts the relationships of species in a tree format. Phylogenetic trees can be either rooted or unrooted. Figure 6.1 illustrates an example of unrooted tree whereas Figure 6.2 illustrates an example of rooted tree. In general, phylogenetic trees are composed of branches, terminal nodes, and internal nodes. An unrooted tree can be rooted simply by picking up one of branches on it. The tree in Figure 6.2 is transferred by picking up the branch (pointed by dashed line) in Figure 6.1. Terminal nodes are also called as Operational Taxonomic Units (OTUs). OTUs stand for existing species, which

are *A, B, C, D* and *E* as illustrated respectively in the figures. Internal nodes are called Hypothetical Taxonomic Units (HTUs), which represent common ancestors of OTUs. A special internal node is the common ancestor of all OTUs, and is named the root node. There is no root node for an unrooted tree, thus the direction of evolution process will not be known.

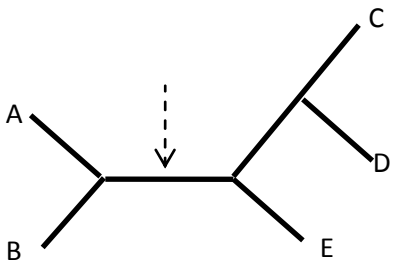


Figure 6.1: An Unrooted Phylogenetic tree

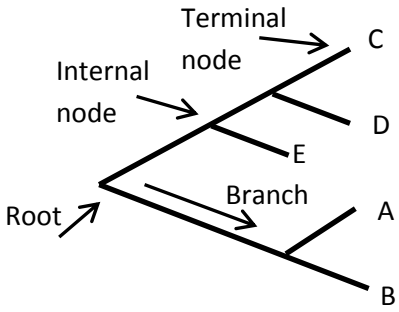


Figure 6.2: A Rooted Phylogenetic tree

As described in chapter 5, distance-matrix based method can be used to construct a phylogenetic tree. Through the method, all sequences are pairwise aligned separately and are combined with their most closest to generate a guide tree. Since distance-matrix based methods do not search all possible tree samples, they achieve less accuracy, but are fast in achieving a search. In this chapter, we will present character-state based method for phylogenetic analysis. Character-state regards each position in the aligned sequence as a character and all possible characters at that position as states. More precisely, for nucleic acid sequences, there are totally 4 character states, which are *A, T, C* and *G* respectively. Each column is compared independently while all characters on that column are compared separately. There are two main advantages of the method. At first, we can reconstruct the character-state of internal nodes to represent ancestral taxa. On the other hand, instead of producing only one tree by distance-matrix based method, character-state based method can afford multiple optimal trees, which is helpful to explore other evolutionary relationships. More details about the character-state based

algorithm, the maximum likelihood algorithm, will be presented in sub section 6.2.2. Table 6.1 lists phylogenetic tree construction methods which are classified by the strategy they use. Distance-matrix based methods utilize stepwise clustering to construct the best tree while character-state based methods search exhaustively the tree space to find the best tree.

Table 6.1: Classified Phylogenetic Construction and Analysis Method

	Strategies	Algorithms
Character State	Exhaustive Search	Maximum Parsimony[4] Maximum Likelihood[3]
Distance Matrix	Stepwise Clustering	UPGMA[2] Neighbour-Joining[1]

In this chapter, the maximum likelihood method was employed to measure the similarity of sample trees in the tree space. The latter is composed by all theoretically possible tree topologies. According to the number of taxa, the number of rooted trees and unrooted trees are given by Equation 6.1 and 6.2 respectively.

$$T(t) = \prod_{i=2}^{t}(2i-3); \quad (6.1)$$

$$T(t) = \prod_{i=3}^{t}(2i-5); \quad (6.2)$$

As shown by Equation 6.1 and 6.2, the number of possible phylogenetic unrooted or rooted tree grows in a factorial way with the number of taxa. Table 6.2 lists the number of possible unrooted trees for up to 12 taxa. While Equation 6.2 might seem simple, but there are 654729075 possible tree topologies for only 12 taxa. Even if we could use a computer to perform the computation of one million trees per second, the work would still need 11

minutes approximately to be completed. If we only increase one more taxa, the time need be extended to 231 minutes. Therefore, accelerations of such problem normally would not make too much sense as it does not scale well. For instance, if we have speeded up the process one thousand times, only increasing three new taxa will immediately offset the speedup effect. Hence such problem is prohibitive even for super computers. Some strategies were put forward to decrease the computation complexity and this is what we will discuss in section 6.2.3 after the introduction of the maximum likelihood algorithm.

Table 6.2: Number of Possible Unrooted Trees For Up To 12 Taxa

Number of Taxa	Number of Unrooted Tree
3	1
4	3
5	15
6	105
7	945
8	10395
9	135135
10	2027025
11	34459425
12	654729075

6.2.2 The Maximum Likelihood Algorithm

Maximum Likelihood (ML) algorithm evaluates a hypothesis about its evolutionary history in terms of the probability that the proposed model and the hypothesized history would give rise to the observed sequence set. The supposition is that a history with a higher probability of reaching the observed state is preferred to a history with a lower probability. ML can be used to infer phylogenetic tree or evolutionary tree, as it searches for the tree with the highest probability or the maximum likelihood from an existing tree. For this,

pairs of biological sequences need to be multiply-aligned in advance. There are many tools designed to perform multiple sequence alignments, such as PAUP [6], clusterW [7]. For a set of sequences with *T* taxa, a rooted tree constructed by them has *T* terminal nodes and *T-1* internal nodes. Suppose there are totally *N* characters in the alignments, the likelihood probability for any arbitrary nodes can be calculated by Equation 6.3.

$$CLP = \prod_{s=1}^{N} T^L(s)T^R(s); \quad (6.3)$$

Where *s* denotes the index of each site in the alignments, T^L and T^R is transition probabilities if the node's descendents are terminal nodes, otherwise conditional likelihood probabilities if the descendents are internal nodes.

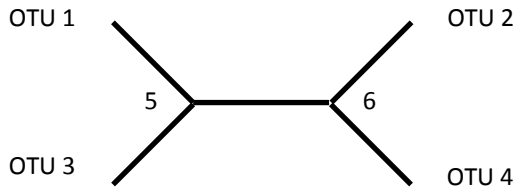
Equation 6.4 presents the computation of likelihood probability for internal nodes where both of their descendents are terminal node. *M* indicates the number of possible ancestor states. Equation 6.5 presents the computation of likelihood probability for internal nodes where both of their descendents are other internal nodes.

$$CLP = \prod_{s=1}^{N} \sum_{t=1}^{M} Tr[S_t \to D_s^L] Tr[S_t \to D_s^R]; \quad (6.4)$$

$$CLP = \prod_{s=1}^{N} \sum_{t=1}^{M} \left(\sum_{j=1}^{M} Tr[S_t \to S_j] CLP_L[s][S_j] \sum_{j=1}^{M} Tr[S_t \to S_j] CLP_R[s][S_j] \right); \quad (6.5)$$

More precisely, suppose there is a set of four aligned nucleotide sequences as our operational taxonomic units as shown below. Suppose also that the tree topology whose likelihood we want to evaluate is as follows:

SITE	1	2	3	4	5	6	7	……
OTU1	A	A	G	A	C	T	T	……
OTU2	C	G	C	C	C	T	T	……
OTU3	A	G	A	T	A	T	C	……
OTU4	G	G	A	G	G	T	C	……



Under the assumption that each nucleotide site evolve independently, the overall likelihood can be multiplied by the likelihood of the first site to the likelihood of the final site. To calculate the likelihood of the 1st site, we need to consider all the possible states occurring in the internal nodes (5, 6) that can be evolved from terminal nodes (1, 2, 3, 4). Given nucleotide sequences, 4 possible substitutions *A, T, C, G* occupy node 5 and 6, so there are 4×4 = 16 possibilities. Therefore, the likelihood of the 1st site can be calculated as:

$$L(1) =$$
$$P(5=A, 6=A) + P(5=A, 6=C) + P(5=A, 6=G) + P(5=A, 6=T) +$$
$$P(5=C, 6=A) + P(5=C, 6=C) + P(5=C, 6=G) + P(5=C, 6=T) +$$
$$P(5=G, 6=A) + P(5=G, 6=C) + P(5=G, 6=G) + P(5=G, 6=T) +$$
$$P(5=T, 6=A) + P(5=T, 6=C) + P(5=T, 6=G) + P(5=T, 6=T) ;$$

Where, $P(5=A, 6=A)$ is equal to $Tr[A->A]Tr[A->A] \times Tr[A->C]Tr[A->G]$. The latter gives evolutionary likelihood of the corresponding replacement between gene characters. The likelihood for the full tree then is the product of the likelihood at each site which can be expressed as:

$$L = L(1) \times L(2) \times \ldots\ldots \times L(N) = \prod_{i=1}^{N} L(i)$$

Since the likelihood of each individual site is extremely small, normally it is expressed as log likelihood. Thus the log likelihood of the entire tree topology can be rewritten as:

$$lnL = lnL(1) + lnL(2) + \ldots\ldots + lnL(N) = \sum_{i=1}^{N} lnL(i)$$

So far, we have presented the method to compute likelihood probability for tree topologies. Since the number of tree topologies grows in a factorial way, exhaustive search of all possible trees is prohibitive even for high performance hardware acceleration platform [8]. In next section, we will present a strategy which is widely used to decrease the computation complexity in phylogenetic analysis.

6.2.3 Bayes Theorem

Bayes theorem [9] is a theorem of probability theory firstly stated by the Reverend Thomas Bayes. It can be seen as a way to estimate the probability of an insight occurring when incomplete information is provided. The probability of a theory normally can be corrected by new evidences through Bayesian probability formula. The fundamental idea of Bayesian statistical theory is to get a known conditional probability parameter and prior probability, then use the Bayes formula to calculate the posterior probability, and finally affect the probability of the insight according to the posterior probability. Bayes theorem has been widely used in many areas [10][11][12].

Suppose $\theta 1, \theta 2, \ldots, \theta n$ are N possible prerequisites or hypotheses that we want to test, then $P(\theta i)$ represents the probability estimated for each prerequisite in advance. The sum value of all $P(\theta i)$ should be equal to 1. The probability stands

for people's degree or belief, or subjective judgements that supporting θi to be true. In particular, $P(\theta i)$ presents our best estimate of the theory we are considering, prior to consideration of any new pieces of evidence. It is known as the prior probability of θi. The prior probability distribution represents the underlying probability distribution, it is not precise sometimes as it is difficult to accurately sample a random variable distribution including missing data to confirm or disconfirm an insight.

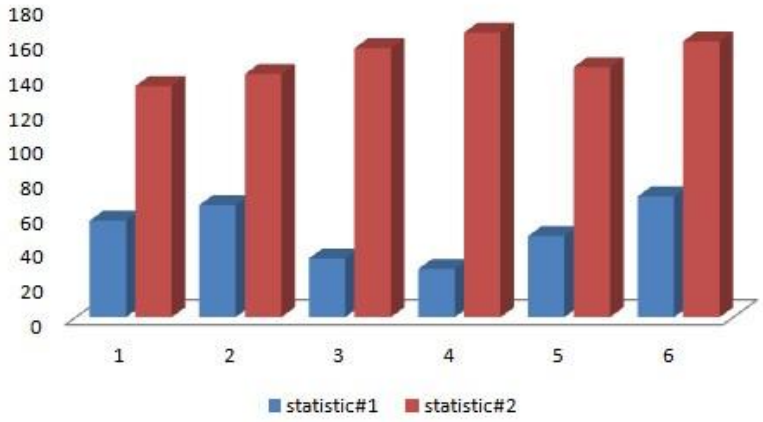


Figure 6.3: The statistics of dice throw; statistic#1 has 300 tries while statistic #2 has 900 tries.

Figure 6.3 illustrates the statistics of dice throw, where, x-axis denotes the point number and Y-axis denotes the number of occurrence. Theoretically, the distribution of dice value should satisfy a uniform distribution, i.e. all outcomes are equally probable. However, if there are relatively few tries, the result turns out to be much different from the theoretical value as illustrated in statistic 1, which can be treated as the prior probability for a particular point. statistic 2 is much closer to the theory probability after increasing the number of dice throws. Therefore, if there is data Di discovered in the process which is based on the truth of θi, then a new evaluation parameter $P(Di|\theta i)$ can be obtained. This is conditional probability that an insight is true given that another insight is true. More precisely, suppose there are two boxes of biscuits, the name of boxes are A and B respectively. A has 10 chocolate flavour biscuits

and 30 plain flavour biscuits, while *B* has 20 of each. Since we choose one of the two boxed without bias, the prior probability of choosing either box is equal to 0.5. The conditional probability of picking a chocolate biscuits can be calculated as 10 / 40 = 0.25 on the observation of *A*. Based on the prior and the conditional probabilities, we can form a full probability model as:

$$P(D, \theta) = P(D|\theta) \times P(\theta);$$

Using the idea of conditional probability, the posterior probability of θi can be set as $P(\theta i | Di)$, which is the probability that θi is true given that Di is true. With the above, Bayes formula can be expressed as below:

$$P(\theta|D) = \frac{P(D, \theta)}{P(D)} = \frac{P(D|\theta) \times P(\theta)}{P(D)}; \quad (6.6)$$

where $P(D) = \sum_{i=1}^{N} P(D|\theta i) \times P(\theta i) = \int P(D, \theta) d\theta$, N are all possible hypotheses. Bayes formula can be reformulated in the continuous domain.

$$P(\theta|D) = \frac{P(D|\theta) \times P(\theta)}{\int P(D, \theta) d\theta};$$

Therefore, if the biscuit randomly picked turns out to be a chocolate flavour biscuit, the probability that the biscuit is picked from *A* can be computed as:

$$P(A|chocolate)$$

$$= \frac{P(chocolate|A) \times P(A)}{P(chocolate|A)P(A) + P(chocolate|B)P(A)}$$

$$= \frac{0.25 \times 0.5}{0.25 \times 0.5 + 0.5 \times 0.5} = 0.33$$

Before we observed the biscuit, the probability we have chosen *A* was the prior probability, *P(A)*, which was 0.5. After observing the biscuit, we need to revise the prior probability to *P(A | chocolate)*, which is 0.33. The result shows that the probability that we chose *A* decreased. Note that only two hypotheses exist in this case, the exact computation of $\int P(D,\theta)d\theta$ is typically analytically intractable for real world applications however.

6.2.4 Monte Carlo Integration

Suppose we wish to compute a complex integral *h(x)* in space *T* as below:

$$\int_T h(x)dx\,;$$

Let *f(x)* be an arbitrary data function and express it as $D_1, D_2, \ldots, D_N$. Let *p(x)* be a probability density function correlating the chance of data occurrence, and express it as $P_1, P_2, \ldots, P_N$. If we can decompose *h(x)* into the product of the function *f(x)* and the probability density function *p(x)* in the space *T*, then the integral can be expressed as an expectation of *f(x)* over the density *p(x)*.

$$\int_T f(x)p(x)dx = Ep(x)[f(x)]; \quad (6.7)$$

Suppose there are totally *N* random variables in the space *T*, then *Ep(x)* can be expressed as below:

$$Ep(x) = \frac{P_1 + P_2 + \cdots + P_n}{N} = \frac{1}{N}$$

So the Equation 6.7 can be written as:

$$\int_T f(x)p(x)dx = \frac{1}{N}\sum_{1}^{N} \mathrm{f(x)} \quad (6.8)$$

Therefore, if we can sample considerable number of random variables from *T*, then the integral *h(x)* can be approximated by Equation 6.8. This is referred to as Monte Carlo integration. As discussed before, the denominator of Bayesian formula is sometimes of multiple dimensions, often unknown and too difficult to compute. Monte Carlo integration can be used to approximate the posterior distributions for Bayesian analysis. We can construct various point estimators of θ so that the mean of posterior distribution *f(θ, D)* can be presented as:

$$E(f(\theta, D)) = \int f(\theta)P(\theta|D)d\theta = \frac{\int f(\theta)\, P(D|\theta)P(\theta)d\theta}{\int P(D,\theta)d\theta}$$

where *p(θ)* and *f(θ)* are the prior destiny distribution and posterior destiny distribution respectively. Let Ω denote the sample space, distribution *f(θ, D)* is arisen from distribution Ω*(θ, D)*. Assuming that the probability of the entire sample space is 1, we can get that the mean of posterior distribution is proportional to $\int f(\theta)\, \Omega(\theta, D)d\theta$. Thus, if we can precisely sample from Ω, then the mean of posterior distribution can be expressed as below:

$$E\big(f(\theta, D)\big) \approx \frac{1}{n}\sum_{1}^{n} f(\theta i, Di)$$

This formula relies on large number of samples so that $E(f(\theta, D))$ can be approximated. However, a major problem towards more widespread implementation of Bayesian approach is that obtaining the posterior distribution often requires the integration of high-dimensional functions which can be computationally difficult as Ω could be very complicated. To overcome this problem, we turn to the concept of Markov Chains. Markov Chain Monte Carlo (MCMC) offers a way of numerically approximating this quantity.

6.2.5 Markov Chain Monte Carlo

Markov Chain is a sequence of random samples, which we denote as S_0, S_1 … S_t. It uses the previous sample value to randomly generate the next sample value so that the current state of S_t depends only on its previous state S_{t-1} in Markov process as below:

$$Pr\ (S_t \mid S_0, S_1 \ldots S_{t-1}) = Pr\ (S_t \mid S_{t-1});$$

If most of the samples in the chain follow stationary distribution, then we can use these samples to perform Monte Carlo integration, the so called Markov Chain Monte Carlo. Normally, the first state of the chain is initialized with some prior values, which may be very different with values of samples from final stationary distribution. After a sufficient long period of running, the states of the chain approaches its stationary distribution, this process is called burn-in. As shown in Figure 6.4 below, the chain seems converge from the 150^{th} samples.

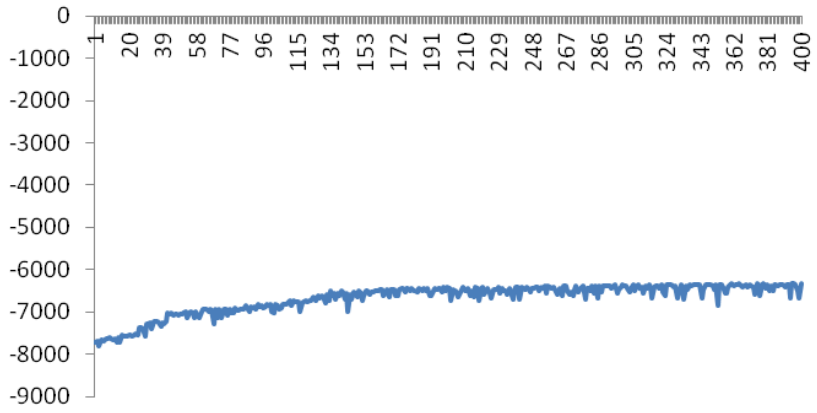


Figure 6.4: An Example of Probability Convergence in Markov Chain

As we discussed above, the current state in Markov Chain depends only upon its previous state and it then randomly generates the next state value. The problem now is how to select the randomly generated states to construct a chain with stationary distribution. In phylogenetic analysis, the chain

randomly constructs an initial tree topology and continues to make moves on tree topologies until the likelihood probabilities of tree topologies satisfy a stationary distribution. For how to make the samples fit stationary distribution, we will refer to Metropolis-Hastings Algorithm in the next sub section.

6.2.6 Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm [13][14] aims to generate a sequence of samples to simulate distribution based on MCMC method. Suppose the variables sampled from distribution $\Omega(\theta, D)$ satisfies that $\Omega(\theta, D) = f(\theta, D)/K$, the algorithm can be described as below:

1. Start with any initial θ satisfying $f(\theta) > 0$.
2. Sample a candidate θ^*, make move on the current point θ_t, then compute the density of the candidate point θ^*.
3. Given the candidate point θ^*, calculate the ratio of density at the candidate point and the current point.

$$\mu = \frac{\Omega(\theta^*, D)}{\Omega(\theta_t, D)} = \frac{f(\theta^*, D)}{f(\theta_t, D)}$$

4. If the candidate point θ^* increases the density ($\mu > 1$), accept the candidate point and set $\theta_t = \theta^*$, then return to step 2. Otherwise, Markov Chain accepts it with probability μ and return to step 2.

Hasting generalized the Metropolis algorithm by using an arbitrary transition probability function $Tr(\theta^*, \theta_t)$, and setting the acceptance probability for a candidate point. This is the Metropolis-Hastings algorithm which is frequently used in computational physics, computational biology, econometrics, and a variety of other fields.

$$\mu = min\ (1,\quad \frac{\Omega(\theta^*, D)Tr(\theta^*, \theta t)}{\Omega(\theta_t, D)Tr(\theta t, \theta^*)})$$

In this section, we presented the essential background on Bayes theorem, the relationship between prior probability and posterior probability and its transition formula. Moreover, we presented the principle of Markov Chain and Monte Carlo integration. The former makes samples from targeted distributions and the latter aims to utilize these samples to simulate complex integrals. In the next section, we will first present the GPP-based phylogenetic analysis software, namely MrBayes, and then present our GPU-based multi-threaded design and implementation of it.

6.3 Phylogenetic Analysis on MrBayes Software

Since the proposed implementation is based on the framework of the most widely used software, namely MrBayes, we first present how the most inner procedures work before reporting the details of our implementation. As illustrated in Figure 6.5, each Markov Chain constructs a tree topology randomly from tree space and stores it in its own space. The initial tree is then pushed into the process of computing the likelihood probability. Afterwards, the move procedure is performed on each tree topology which is done simply by changing the branch order on the tree. For instance, delete *T1* and move it to the branch between *T4* and *N1* in Figure 6.7. The metropolis-Hastings algorithm is utilized to compare the likelihood probabilities to decide whether to accept the new tree topology or not. If the move increases the likelihood probability, the following move will be made on the new modified tree topology. Otherwise, the previous tree topology is kept for the next iteration.

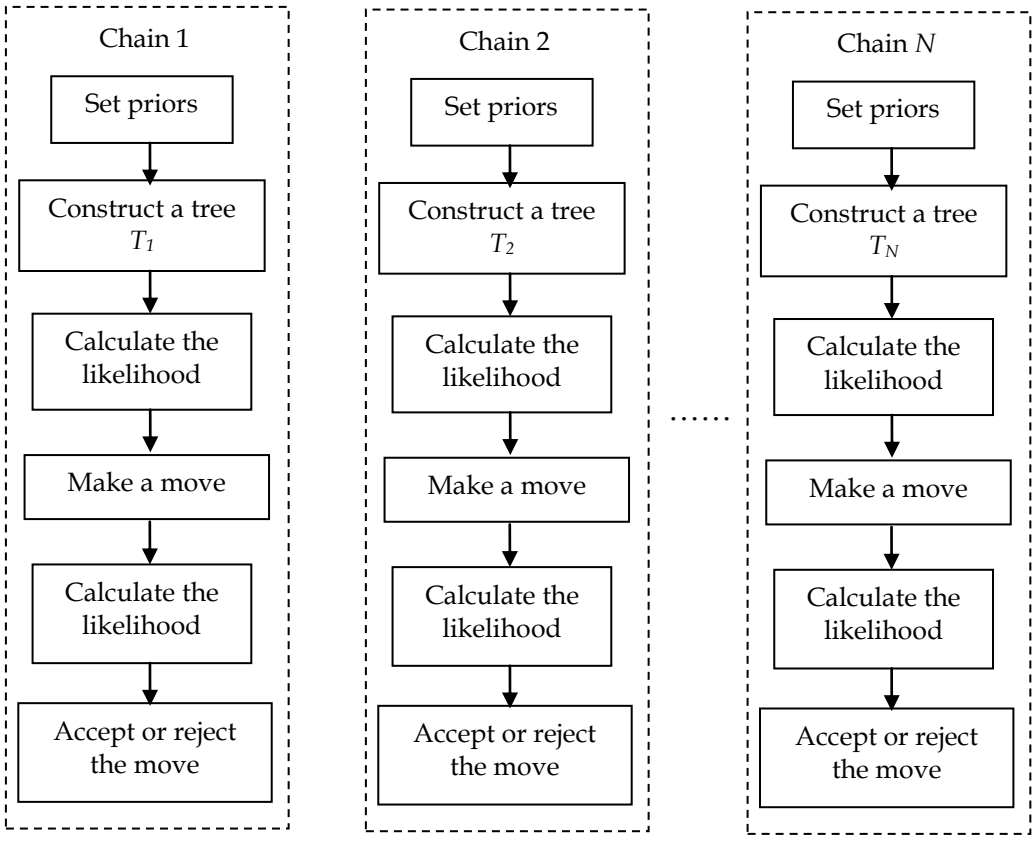


Figure 6.5: The most inner procedures under the framework of MrBayes software, the swap of chains should be performed after making the decision of accept or reject the move on the tree of each chain.

6.3.1 Multiple Chains

In MrBayes software, multiple chains normally exist in one run. The motivation is to allow multiple peaks in the landscape of trees to be more readily explored as standard MCMC implementation may be prone to entrapment in local optima. Multiple chains are scheduled serially from the first one to the last one. Each chain has its own initial tree topology and the computation of relevant likelihood probability is independent of each other. Figure 6.6 presents the implementation of making a move on the tree topology, where *CLPs* stand for the Conditional Likelihood Probabilities. Once all chains complete a round, the chain with tree topology having maximum likelihood

probability is set as 'cold' chain and the tree topology it has is then distributed to other chains. This method is called as Metropolis Coupled MCMC [15]. The number of generations in each chain stands for the number of tree samples, which normally depends on the convergence degree of likelihood probabilities.

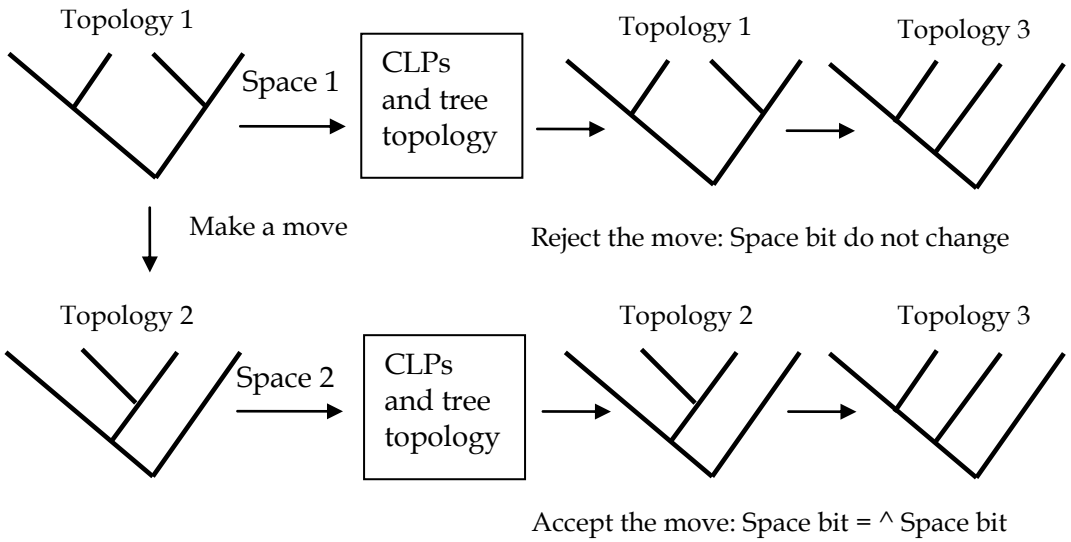


Figure 6.6: Making move on tree topology, each chain has two blocks of space for current tree and future tree.

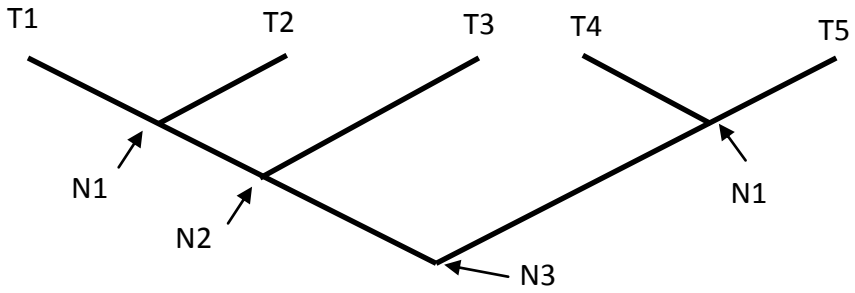


Figure 6.7: A rooted tree topology and its three alternative nodes

6.3.2 The Computation of Likelihood

Let $C = (C_1, C_2, \ldots, C_n)$ represent N aligned columns, let $D_c = (D_{c1}, D_{c2}, \ldots, D_{ci})$ represent all the characters on column c. Suppose that each character is evolved in one of k possible states which are labelled as $\{S_1, \ldots, S_k\}$. Given M

species, there are totally *M* terminal nodes, *M-2* interior nodes and 1 root node. As illustrated in Figure 6.7, each interior node can be summarised as one of the following three alternatives:

- Case 1: *N1* (terminal node and terminal node)
- Case 2: *N2* (terminal node and interior node)
- Case 3: *N3* (interior node and interior node)

Let *CLP* denote the conditional likelihood probability, CLP_L and CLP_R denote the conditional likelihood probability of its left and right descendent, respectively. *z, i* and *j* denote the evolutionary state of interior nodes and their descendent respectively. Let *Tr* denote the transition probability matrix. Since the evolutionary model used in the proposed approach is for amino acid sequences, so that for case 1, if *C=1*, *N1* can be calculated as follows:

$$CLP(S_1, \dots, S_k) = \sum_{z=1}^{k} Tr[S_z \rightarrow D_1^L] Tr[S_z \rightarrow D_1^R];$$

For alternative 2, *N2* can be calculated as follows:

$$CLP(S_1, \dots, S_k) = \sum_{z=1}^{k} \sum_{j=1}^{k} Tr[S_z \rightarrow S_j] CLP_L[S_j] Tr[S_z \rightarrow D_1^R];$$

For alternative 3, *N3* can be calculated as follows:

$$CLP(S_1, \dots, S_k) = \sum_{z=1}^{k} \sum_{j=1}^{k} Tr[S_z \rightarrow S_j] CLP_L[S_j] \sum_{i=1}^{k} Tr[S_z \rightarrow S_i] CLP_R[S_i];$$

The calculation of all unobserved nodes can be summarised as the following equation:

$$CLP = \prod_{0}^{N} \sum_{z=1}^{k} \sum_{j=1}^{k} Tr[S_z \rightarrow S_j] CLP_{(L+N)}[S_j] \sum_{i=1}^{k} Tr[S_z \rightarrow S_i] CLP_{(R+N)}[S_i];$$

Where, $CLP_{(L+N)}$ and $CLP_{(R+N)}$ stand for the conditional likelihood probabilities of the N^{th} aligned columns. Figure 6.8 presents the pseudo code of computing likelihood probability of internal nodes in MrBayes software.

```
Pointer *TrL, *TrL, *Tree;
scalar h, c, i, j;
TrL ← TransitionProbability;
TrR ← TransitionProbability;
lState ← Tree.left;
rState ← Tree.right;
h ← 0;

For c ← 0 to numChars
{
        i = lState[c];
        j = rState[c];
        For k ← 0 to numGammaCats
        {
                CLP[h++] ← TrL[i++] * TrR[j++];
                CLP[h++] ← TrL[i++] * TrR[j++];
                CLP[h++] ← TrL[i++] * TrR[j++];
                CLP[h++] ← TrL[i++] * TrR[j++];

                i += 16;
                j += 16;
        }
}
```

Figure 6.8: Pseudo code of GPP-based implementation of computing likelihood probability for internal nodes

6.4 GPU-based Multi-threaded Design and Implementation

6.4.1 The Parallelization of Running Chains

As discussed above, chains for computing likelihood probabilities are scheduled serially in MrBayes GPP-based implementation. The elapsed time grows linearly proportional to the number of local chains. Base on the truths that the number of unobserved tree nodes is constant in arbitrary tree topology and each chain computes independently, the process can be well performed in parallel. As illustrated in Figure 6.9, The proposed GPU-based method distributes an equal amount of threads to individual running chain, which is performed simply by setting the dimension of thread block to (16, 4) and the

dimension of grid to (7, *numLocalChains*). The total threads are partitioned by *blockIdx.y* ranging from 0 to *numLocalChains* – 1. More precisely, the corresponding thread blocks to each running chain are indexed by *blockIdx.y*. Therefore, the total number of threads for each chain is equal to 448 (16×4×7). The grid width is configurable and depends on the number of sites in multiple alignments and GPU hardware resources. In the proposed implementation, it is equal to 7. The threads allocated for chains are then applied to corresponding sites.

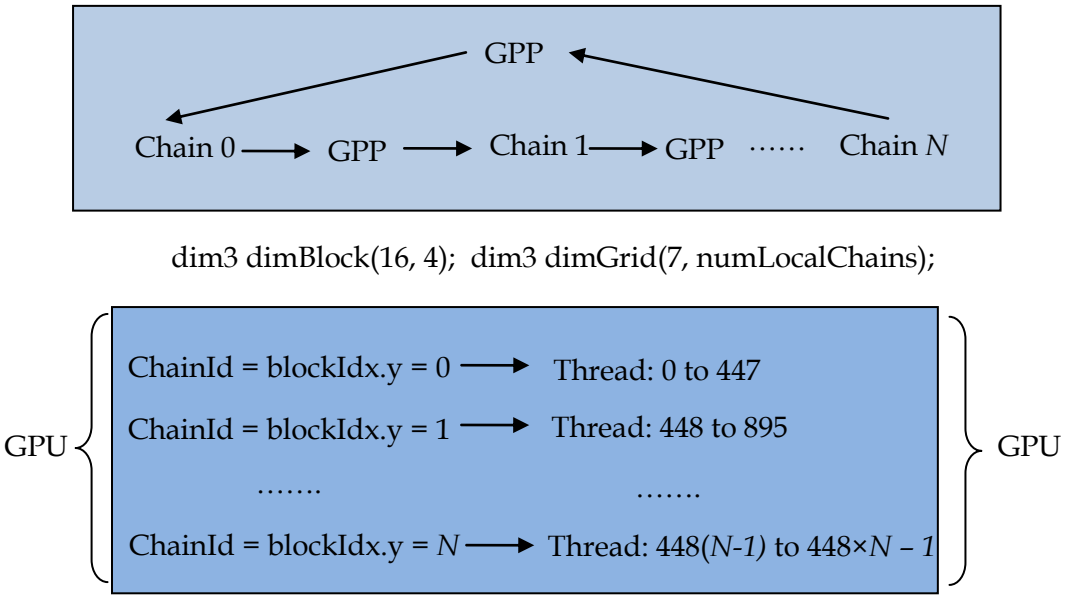


Figure 6.9: The architecture of parallelizing multiple chains on GPU

6.4.2 The Parallelization of Likelihood Computation

The likelihoods-based phylogenetic analysis software MrBayes exploits the fact that the computation of internal nodes depends only on its immediate children, so it uses a post-order traversal of all the internal nodes until the computation reaches the root of the tree topology. This approach was presented by Felsenstein [3]. It computes all internal nodes recursively by using all conditional likelihood probabilities of their descendent nodes, and then sums all the likelihoods of all aligned characters to produce the likelihoods of a

given tree. Suppose there are *N* aligned biological sequences with length *L*, the evolutionary model is set to four by four nucleotide model with gamma-distributed rate variation across sites. The number of computational elements of the internal node with two descendent internal nodes can be presented as:

$$\begin{aligned} CLP(D) &= CLP(D_1) \times numOfSites \\ &= GammaCategories \times states \times numOfSites \\ &= 16 \times numOfSites \end{aligned}$$

The substitution model is set to the widely used General Time Reversible (GTR) model, which has six substitution types, one for each pair of nucleotide. The computation of conditional likelihood probabilities for each internal node in MrBayes phylogenetic analysis approach is performed serially which results in a high computation cost. In our implementation, we exploit the parallelism potential and distribute the aligned characters across threads. The life time of each thread can be approximated as the ratio of total conditional likelihoods and the number of threads for the chain. Pseudo code in Figure 6.10 outlines our implementation of Equation 6.5. By default, MrBayes software implements phylogenetic analysis by supposing all characters evolve at the same rates. However, the assumption that the rate of nucleotide substitution is constant over different nucleotide sites is sometimes unrealistic as residues in real sequence data may have different functional constraints [16][17][18]. Given an internal node, with two other internal nodes as its descendents, there are 4 conditional likelihood elements representing each site if the rate of variation over sites is constant and 16 conditional likelihood elements if the rate of variation over sites is gamma-distributed. Since there are 4 gamma rates corresponding to 4 substitution matrixes in gamma-distributed model, we separate threads in each individual block into 4 groups to calculate the relevant proportion of conditional likelihoods which is simply implemented by using *threadIdx.y*. As illustrated in Figure 6.11, 64 threads fetch 16 characters by

Define dimBlock(t_x, t_y) as dimBlock(16, 4) to specify thread hierarchy;
Use x to specify number of columns;
Use y to specify number of gamma category;

1: Define number of columns processed by thread as COLUMN_THREAD (CT);
2: Define threads for relevant gamma category as GAMMA_THREAD (GT);
3: for c = t_x to t_x×CT, h = 16×c, all threads do in parallel:
4: $\Omega_L = \sum_{S1=A}^{T}(\sum_{S2=A}^{T} Tr_{GT}[S1S2] \times CLP_L[S2])$;
5: $\Omega_R = \sum_{S1=A}^{T}(\sum_{S2=A}^{T} Tr_{GT}[S1S2] \times CLP_R[S2])$;
6: $\Omega(GT, h) = \Omega_L \times \Omega_R$;
7: $CLP_L += 256$;
8: $CLP_R += 256$;
9: end for;

Figure 6.10: Pseudo code of GPU-based implementation of computing likelihood probabilities for internal nodes

using *threadIdx.x* ranging from 0 to 15 in each batch. All likelihood probabilities are then stored in GPU global memory.

6.5 Performance Evaluation

In this section, we present the experimental results of the proposed GPU-based phylogenetic analysis implementation compared to the GPP-based MrBayes software. The experimental performance is evaluated on an Nvidia GeForce 460 GTX GPU, while MrBayes software (MrBayes 3.1.2) was implemented on a Dell computer with one Intel Xeon 2.53GHz processor and 6 GB memory installed. The aligned DNA sequences are from the database in MrBayes software [5]. Table 6.3 and 6.4 present the comparative results between the proposed method on GPU and GPP-based MrBayes software with 4 and 8 running chains respectively. It is difficult to precisely determine when terminating a running chain as precisely determining sufficient burn-in samples is difficult to achieve for non-biologist in practice. Therefore, we do not perform the analysis on

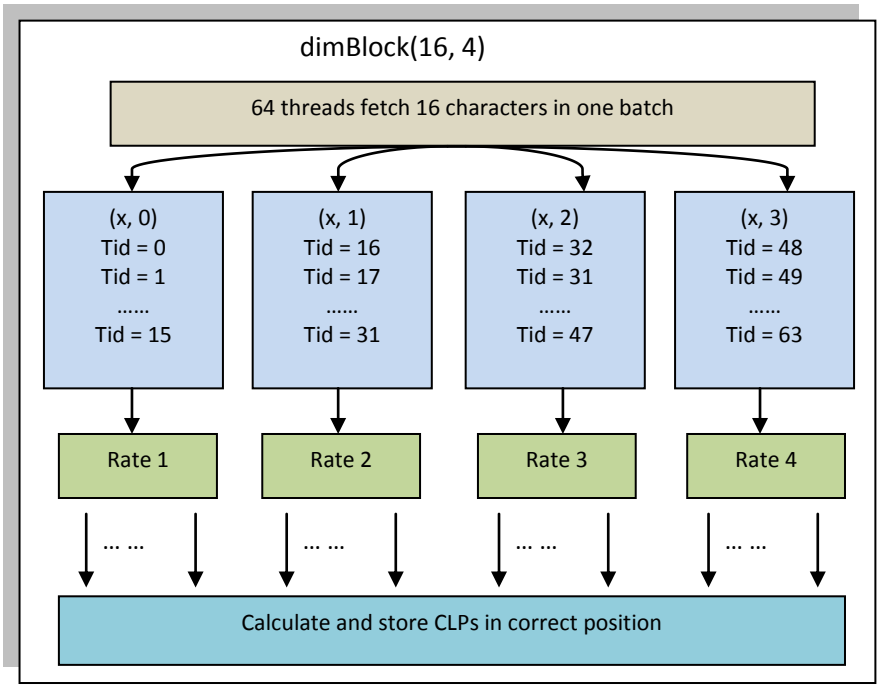


Figure 6.11: The architecture of GPU-based implementation for the computation of likelihood probabilities

determining how many samples we need to take. Instead, we make five groups of performance comparisons from 1000 to 16000 generations. The speedup factors for each case are relatively stable as shown in Table 6.3 and 6.4 and the proposal distribution of likelihood states can be considered convergent after running 1000 generations. The tables show approximate 6x – 8x speedup improvement compared with the optimized GPP-based phylogenetic analysis software with 4 and 8 running chains respectively.

6.6 Conclusions and Future Work

In this chapter, we presented a GPU-based multi-threaded design and implementation of phylogenetic analysis. The proposed method is based on the framework of MrBayes software, which utilizes Markov Chains to sample trees and maximum likelihood method to compute tree probabilities. Through tree selection mechanism, the optimal tree is finally obtained after the proposal

Table 6.3: Performance comparison between MrBayes software and the proposed GPU implementation with 4 running chains

Gen[*]	*Xeon*	*GTX460*	*Speedup*	*State*[**]
1000	1.704	0.304	5.60	-6291.52
2000	3.420	0.565	6.05	-6287.57
4000	6.851	1.145	5.98	-6286.08
8000	13.70	2.277	6.01	-6284.76
16000	27.47	4.680	5.87	-6285.22

Table 6.4: Performance comparison between MrBayes software and the proposed GPU implementation with 8 running chains

Gen[*]	*Xeon*	*GTX460*	*Speedup*	*State*[**]
1000	3.418	0.421	8.11	-6289.84
2000	6.832	0.852	8.02	-6287.33
4000	13.71	1.632	8.40	-6286.06
8000	27.28	3.386	8.05	-6285.10
16000	54.89	6.509	8.43	-6285.59

Gen[*]: The number of generations *State*[**]: The best likelihood probability

distribution of likelihood states converge usually after a relatively long execution time. The idea of parallelizing the application is put forward based on the fact that the time grows linearly with the number of chains so that serial computation chains can be split and processed in parallel. Moreover, the computation of likelihood on internal nodes can be performed in parallel as the computations at different sites on the same tree nodes are independent of each other. The proposed method achieves 6x – 8x speedup factors compared with optimized GPP-based phylogenetic analysis. There are another two areas

which can be potentially improved. The first one is based on the fact that not all internal nodes are dependent of each other. Hence nodes which have no inter dependence can be computed in parallel. Another one is based on the fact that the likelihood probabilities for some nodes is steady after making a move on tree topology, hence redundant computation can be avoided. Future work aims to improve these shortcomings just discussed and scale up the application into multiple GPUs.

6.7 References

[1] Saitou M. and Nei N.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol, 1987, 4:406-425.

[2] Farris J.S.: Estimating phylogenetic trees from distance matrices. American Nature, 1967, 155:279-284.

[3] Felsentein J.: Evolutionary trees from DNA sequnces: a maximum likelihood approach, J.Mol.Evol, 1981, 17:368-376.

[4] Fitch W.M.: Toward defining the course of evolution: Minumum changer for a specfic tree topology. Systematic Zoology, 1971, 20:406-416.

[5] Download website for MrBayes, 16th April 2012 retrieved from http://mrbayes.sourceforge.net/download.php.

[6] Download website for PAUP, 16th April 2012 retrieved from http://paup.csit.fsu.edu/down.html.

[7] Thompson J.D., Higgins D.G. and Gibson T.J.: CLUSTALW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res 1994. 22:4673–4680.

[8] Kasap S. and Benkrid K.: High Performance Phylogenetic Analysis With Maximum Parsimony on Reconfigurable Hardware. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2011, 19(5):796-808.

[9] Thomas B. and Richard P.: An Essay towards solving a Problem in the Doctrine of Chance. Philosophical Transactions of the Royal Society of London, 1763, 53(0):370–418.

[10] Box G.E.P. and Tiao G.C.: Bayesian Inference in Statistical Analysis. Wiley, 1973, ISBN 0-471-57428-7.

[11] Foreman L.A, Smith A.F.M. and Evett I.W.: Bayesian analysis of deoxyribonucleic acid profiling data in forensic identification applications. Journal of the Royal Statistical Society, 1973, Series A, 160:429–469.

[12] Baldi P. and Long A.D.: A Bayesian framework for the analysis of microarray expression data: regularized t-test and statistical inferences of gene changes. Bioinformatics, 2001, 17(6):509-519.

[13] Metropolis N., Rosenbluth A.W. and Rosenbluth M.N.: Equations of State Calculations by Fast Computing Machines.Journal of Chemical Physics, 1953, 21(6):1087–1092.

[14] Hastings W.K.: Monte Carlo Sampling Methods Using Markov Chains and Their Applications". Biometrika, 1970, 57(1):97–109.

[15] Altekar G., Dwarkadas S., Huelsenbeck J.P. and Ronquist F.:Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. Bioinformatics 2004, 20(3):407-415.

[16] Felsenstein J. and Churchill G.A.: A Hidden Markov Model approach to variation among sites in rate of evolution, and the branching order in hominoidea. Molecular Biology and Evolution, 1996, 13(1):93–104.

[17] Tamura K.: Estimation of the number of nucleotide substitutions when there are strong transition-transversion and G+C content biases. Molecular Biology and Evolution, 1992, 9(4):678–687.

[18] Tavaré S.: Some Probabilistic and Statistical Problems in the Analysis of DNA Sequences. American Mathematical Society, 1986, 17:57–86.

Chapter 7
Evaluation of Graphics Processing Units in High Performance Computing

7.1 Introduction

In this chapter, we present an evaluation of the overall work presented in this thesis. We focus on a discussion of different GPU solutions presented in previous chapters as well as lessons learnt from this work in optimizing GPU programs, and a comparative study of GPUs in relation to other computer technologies. In particular, as we presented two different GPU strategies for the widely used Smith-Waterman pair-wise sequence alignment algorithm in chapter 3 and 5, we first evaluate in section 7.2 the different thread use strategies in these implementations. Afterwards, we present in section 7.3 a number of GPU computing optimizations learnt throughout this work including: the efficient use of threads and blocks based on the number of registers and the size of shared memory on GPU, efficient global memory access, efficient shared memory access. After that, we present in section 7.4 an evaluation of GPU technology in high performance BCB, compared to with other technologies, namely Field Programmable Gate Arrays (FPGAs), and General Purpose Processors (GPPs). Comparison criteria include purchase cost, development cost, speed performance, and energy consumption.

7.2 Thread Allocation and Scheduling Strategies

In chapter 3, we presented a GPU-based design and implementation of the Smith-Waterman algorithm using two task parallelization approaches, namely: inter-task and intra-task parallelization approach. In chapter 5, we used another version of the intra-task parallelization approach to perform the first step of Multiple Sequence Alignments (MSAs), which adopts the Smith-

Waterman algorithm to perform a forward and a backward phase on the alignment matrix involved by two sequences. Hence, before we evaluate the two intra-task parallelization approaches, we first give a description of both intra-task approaches. We then present the reason why the first approach cannot be mapped into the second one. A discussion of efficient thread allocation is finally laid out.

The whole process of our proposed approach presented in chapter 3 can be simply described in three steps: 1) Read database sequences and residues in query sequence from global memory and constant memory respectively. 2) Compute the relevant rows in the alignment matrix. 3) Write highest score back into global memory and send to the host. Step 2 can be performed without accessing global memory (to store and read *H* and *F* values) if enough threads are allocated to process the residues in query sequence and there is enough shared memory. The proposed approach was based on the fact that data transfer between global memory and the stream processors (SPs) is much slower than data transfer between shared memory and steam processors, and for each alignment matrix, the query sequence is the same. Therefore, we allocated as many threads as possible to decrease the thread iteration count, thus decreasing the data transfer cost between SPs and global memory. The dimensions of the blocks and grid were set as shown in Table 7.1:

Table 7.1: The partition of kernel of the Smith-Waterman algorithm

Dimension	width	Height
Block	1	x
Grid	1	y

Where x denotes the number of residues in the query sequence, and y denotes the total number of subject sequences. Since threads on GPU device are packed in warps and executed in SIMD model, x is normally set as a multiple of the warp size (recall it is 32 in the GPU used in this work). For instance, if the query length is 127, the x is set to 128 (32×4).

In chapter 5, we proposed another intra-task based approach to perform the implementation of the Smith-Waterman algorithm. Since each pair of sequences is from a distance matrix, we do not have a static query sequence in this case. Hence, allocating a relatively big number of threads would normally lead to thread load imbalance. For instance, suppose there are 4 pairs of sequences, the shortest sequence is 32 while the longest sequence is 128. Threads in warps 2, 3 and 4 allocated for the matrix having the shortest sequence will be idle if we use the aforementioned approach. Therefore, we performed another task allocation strategy given in Table 7.2:

Table 7.2: The partition of kernel for MSAs

Dimensions	width	Height
Block	w	32
Grid	1	z

Where w denotes the number of pairs processed in each thread block, and z denotes the total number of pairs in the distance matrix divided by w. Here we allocate a warp of threads for the computation of each alignment matrix. Given that the maximum number of active warps per Stream Multiprocessor (SM) is 24 and the maximum number of active blocks per SM is 8, w is set to 3 to fulfill the maximum use of hardware resources if we use 8 thread blocks. More information will be discussed in detail in section 7.3.

7.3 GPU Program Optimizations

7.3.1 Resource Allocation

The GeForce 8800 GTX card has 16 SMs and 128 SPs, with each SM having 8192 registers and 16K bytes shared memory for fast data access. The maximum number of active threads per SM that can execute in parallel is 768, leading to a total number of active threads per GPU equal to 12288. However, this would allow each thread to use only a very limited number of registers.

Table 7.3 shows the limited number of registers and size of shared memory to reach the maximum number of active threads.

Table 7.3: Memory as a limiting Factor of Parallelism

	The number of registers	The size of shared memory
active blocks(8)	≤ 1024	≤ 2 Kbytes
active warps(24)	≤ 341	≤682 bytes
active threads(768)	≤10	≤21 bytes

If each thread uses 11 registers, the number of active threads in each SM will be reduced to a theoretical 8192/11 = 744. However, as threads are grouped in blocks and the resource allocation is done at the block granularity, this theoretical number of active threads cannot be reached. Indeed, let us assume, for instance, a kernel executed by 8 blocks, with each block having 96 threads. If each thread uses 11 registers, only 7 blocks can be active at the same time (indeed 8 blocks would result in 11x96x8 = 8448 registers which is high than the available number of registers of 8192). Thus, the number of active threads is 672, rather than the theoretical 744, a reduction of 1/8 of the maximum number of threads that can execute simultaneously in each SM. If the kernel is executed by 3 blocks, with each block having 256 threads, a reduction of 1/3 of the maximum number of threads occurs. This greatly reduces the number of active warps available for scheduling, thus leading to SPs stalling when performing long-latency operations, i.e. global memory access. A usual approach to solve the problem is to use shared memory instead.

In general, the aim of any resource allocation is to maximize the number of active threads or warps at any particular number of time. For instance, assume an application with a fixed number of tasks, e.g.. 768 , and consider three different thread batching alternatives: 768 blocks with 1 warp per block, 384 blocks with 2 warps per block, and 256 blocks with 3 warps per block respectively. Figure 7.1 illustrates the performance for each alternative.

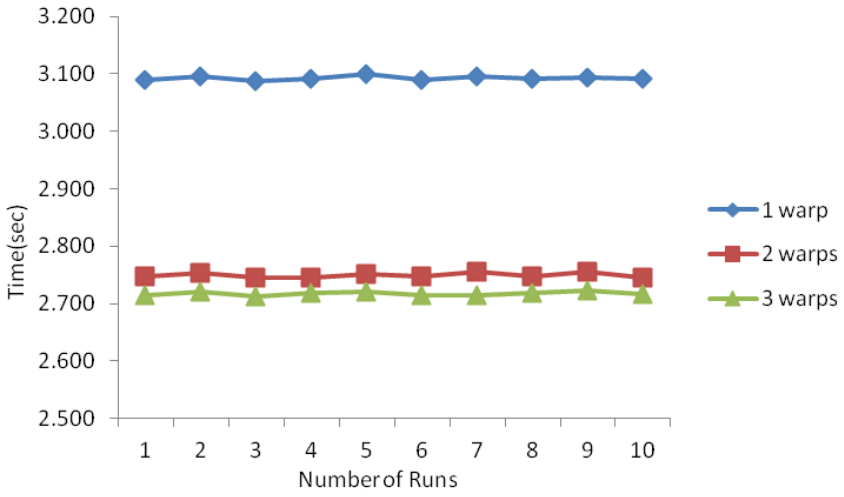


Figure 7.1: Performance comparison between three thread batching methods

Experiments for this evaluation are performed on a simple kernel code in order to achieve the maximum number of active blocks and warps in each SM. Given a fixed number of tasks like in this case, we find that different thread batching approaches result in relatively large performance differences. Indeed, since the maximum number of blocks that can run in parallel in each SM is 8, and the maximum number of warps that can run in parallel in each SM is 24, the first alternative results in 8 active warps at any time, the second results in 16 active warps, and the third results in 24 active warps at any time. Obviously the last two cases achieve better performance as global memory access latency can be hidden better because of the higher number of active threads. We notice that the third alternative outperforms the second alternative with a smaller margin. This is because the number of active threads has nearly reached the saturation threshold in the second alternative (512 out of 768).

7.3.2 Efficient Global Memory Access

The GeForce 8800 GTX card uses 900MHz DDR (double data rate) memory clock and 384-bit wide memory bus. The peak theoretical global memory access bandwidth is calculated as: $((384/8) \times 900 \times 10^6 \times 2)/10^9 = 86.4$GB/sec. Since global memory requests' latency is around 200 clock cycles and 8 instructions can be issued per cycle per SM, the kernel needs 1600 instructions to avoid the stream processors stalling. Suppose the kernel issues 1 global load for every 8 instructions. There are 200 global memory requests before the first request completes. This can easily lead global bandwidth saturation. To reduce the global memory bandwidth usage, we have used two strategies in our experiments. The first strategy is to copy the data from global memory by multiple threads into shared memory iteratively and process the data locally. The second strategy is to make global memory access coalesced. Note that this operation is only needed for devices with compute capability < 1.2. Note also that there is only one memory transaction issue for a half-warp. More details about the principle of coalesced access can be found in chapter 2.

In order to evaluate the improvements that can be gained through coalesced memory access, we made two experiments for each of the above strategies, namely use of shared memory buffering, and coalesced global memory access. The first strategy was used in the computation of alignment matrices in the Smith-Waterman algorithm implementation as explained in Chapter 3 and Chapter 5. Indeed, before computing each cell in the alignment matrix, all residues in a subject sequence are read into shared memory, and the computation is performed by each thread reading residues directly from shared memory, instead of global memory. For a subject sequence of length 32, each thread needs to perform 32 global loads in the initial version. In the improved version, each thread first loads one data from global memory and stores it to shared memory. Then, each thread performs 32 shared memory loads. Figure 7.2 illustrates the performance comparison using both versions.

The experiment shows an improvement of 1.7x speed-up achieved through the use of shared memory as a buffer.

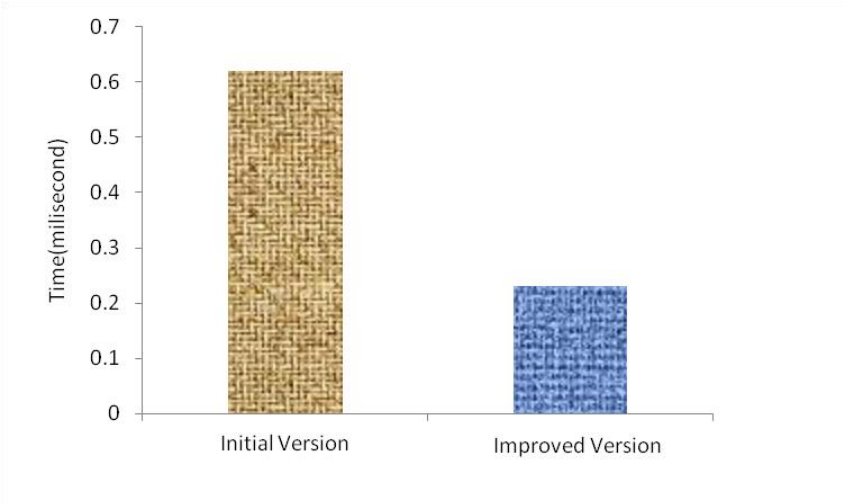


Figure 7.2: Performance comparison after reduce global memory access

Performance comparison between coalesced access and un-coalesced access is illustrated in Figure 7.3. The experiment is performed by a warp of threads for reading data residing in global memory and shows an improvement of 26.2% achieved by coalesced access over un-coalesced access.

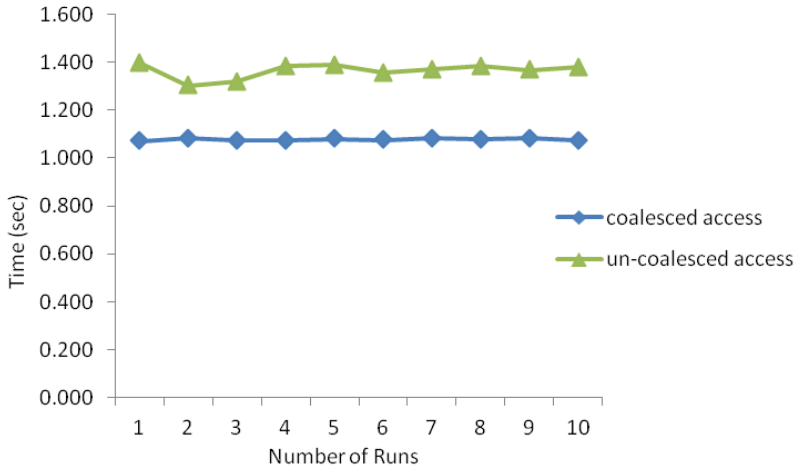


Figure 7.3: Performance evaluation between coalesced and un-coalesced access

The fundamental idea of both strategies is to reduce the global memory transactions issues, and then to decrease the overall execution time.

7.3.3 Efficient Shared Memory Access

The GeForce 8800 GTX card has 16Kbytes of shared memory for each SM, and 256Kbytes of shared memory for the entire device. Access to shared memory is as fast as accessing registers if there is no bank conflict (more details about the architecture of shared memory can be found in chapter 2). In this section, we evaluate five shared memory access approaches using five corresponding shared memory usage equations as follows:

- *shared[0] = data;* *1-way shared memory access*
- *int data = shared[8*tid];* *2-way shared memory access*
- *int data = shared[4*tid];* *4-way shared memory access*
- *int data = shared[2*tid];* *8-way shared memory access*
- *int data = shared[tid];* *16-way shared memory access;*

where *tid* stands for the index of each thread, and *int* denotes an integer variable type (there are another two variables types as will be discussed later in this section). Figure 7.4 illustrates the performance of the above five different shared memory access patterns.

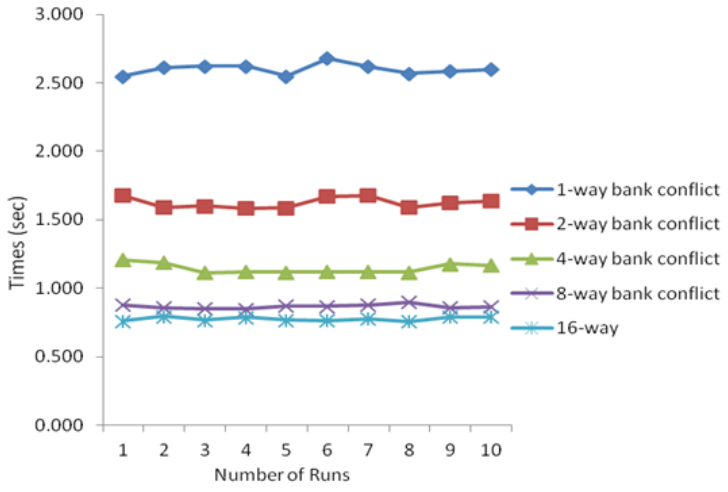


Figure 7.4: Performance comparison of five different shard memory access patterns

From the figure we can see the execution time degrades proportionally to the number of ways. The fastest speed is 16-way shared memory access, which denotes no bank conflict. The lowest speed occurs when all half-warp threads write values to the same address in shared memory, the so called 1-way bank conflict. Since the width of each bank in shared memory is 32-bit, inappropriate use of variable types also leads to bank conflicts. Figure 7.5 illustrates the performance of the following shared memory accesses when using three different variable types:

type data = shared[tid];

Where, *type* is *int*, *unsigned int*, and *char* respectively. The integer type is a 32-bit width and the unsigned integer has 16-bit width. Thus, the latter is equivalent to the 8-way bank conflict of Figure 7.4. Similarly, with the case of character type leads to the 4-way bank conflict, as its word width is 8-bits.

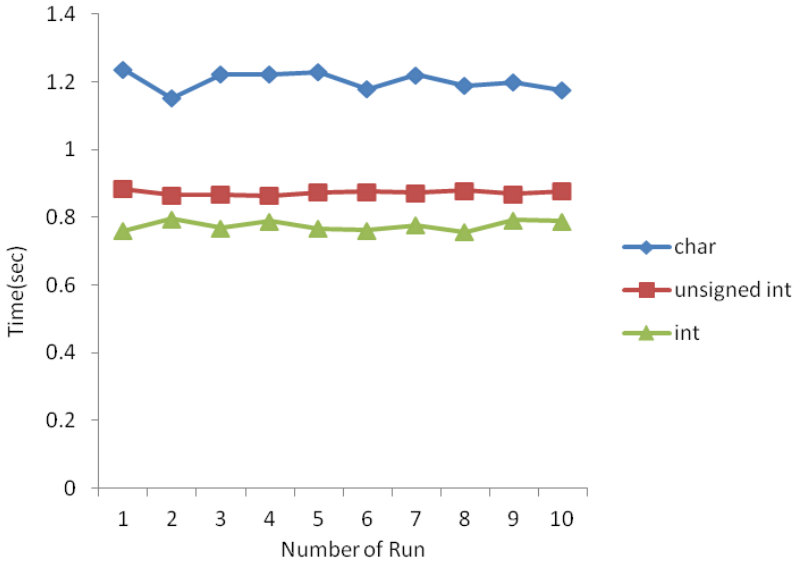


Figure 7.5: Performance comparison of shared memory access by three different variable types

7.4 Comparative Study: GPUs vs. FPGAs vs. GPPs

This section presents the results of a comparative study between three implementation platforms, namely GPU, Field Programmable Gate Arrays (FPGAs), and General Purpose Processor (GPP), in the design and implementation of the Smith-Waterman algorithm [1]. Comparison criteria include purchase cost, speed performance, development cost, and energy consumption. Before reporting the comparison, we briefly introduce the specifications of each platform.

A. The GPU implementation Platform

For the purpose of our GPU-based implementation of the Smith-Waterman algorithm, we targeted a GeForce 8800GTX GPU on a Mac Pro workstation which has two 2.66 GHz Dual-core Intel Xeon 64-bit processor. The GPU was fabricated in 90 nm CMOS technology. All data transfer between the host processor and the GPU pass through the PCI-E ×16 with a bandwidth of 8GB/s.

B. The FPGA Implementation Platform

This implementation was performed by another PhD student in the group, and the details can be found in [2]. The implementation targeted an HP ProLiant DL145 server machine which has an AMD 64-bit processor and a Celoxica RCHTX FPGA board. The latter has a Xilinx Virtex-4 LX160-11 FPGA chip which is based on 90-nm copper CMOS process, which is the same technology of the above GPU implementation to ensure fair comparison. All data transfer between the host processor and FPGA chip on the HP ProLiant server pass through the Hyper-Transport interface with a bandwidth of 3.2 GB/s.

C. The GPP Platform

For the purpose of our GPP-based implementation of the Smith-Waterman algorithm, we targeted a PC with a 3.4GHz Pentium 4 Prescott processor, 1GB of RAM, running Windows XP OS. The Prescott processor has a 31-stage

pipeline, 16K 8-way associative L1 cache and 1MB L2 cache, and like the above two platforms, it is also based on 90nm CMOS technology.

Table 7.4 first presents the execution times of the Smith-Waterman implementation on all three platforms for a number of query sequences against the SWISS-PROT database (as of August 2008) when it contained 392,768 sequences and a total of 141,218,456 characters. If we compare with the query length of 256, this shows the FPGA solution to be two orders of magnitude quicker than the GPP solution (223:1), while GPU solution is one order of magnitude quicker than the GPP solution (14:1). The MCUPS for FPGA, GPU and GPP are equal to 19400, 1200 and 85 respectively.

Table 7.4: Performance comparison between FPGA, GPU and GPP on the implementation of the Smith-Waterman algorithm

Query (Protein name)	Query length	FPGA Time (sec)	GPU Time (sec)	GPP Time (sec)
P36515	4	1.5	4.1	24
P81780	8	1.6	4.1	30
P83511	16	1.6	4.3	43
O19927	32	1.6	4.7	62
A4T9V0	64	1.6	6.7	115
Q2IJ63	128	1.6	12.8	210
P28484	256	1.9	30.0	424
Q1JLB7	512	4.5	76	779
A2Q8L1	768	6.7	136.2	1356
P08715	1024	8.9	172.8	1817

By accounting for the cost of development (measured on the basis of US$20/hour as the average salary of a freshly graduated student where the experiments took place) and the cost of purchase of the respective platforms, Table 7.5 gives the overall development cost of all three solutions. Note here that the purchase cost of the FPGA and GPU platforms includes the cost of the host machine and that the resulting development times for FPGA, GPU and GPP are 300, 45 and 1 in days [1].

Table 7.5: Cost of purchase and development on all three platforms

Platform	Purchase Cost ($)	Development Cost ($)	Overall Cost ($)	Normalized Overall Cost
FPGA	10,000	48,000	58,000	50(58000/1160)
GPU	1450	7,200	8,650	7.5(8650/1160)
GPP	1000	160	1,160	1(1160/1160)

Table 7.5 shows that the FPGA solution is 50x more expensive than the GPP solution, followed by the GPU (7.5x). The performance per dollar spent can thus be calculated by dividing the MCUPS for aligning query sequence of length 256 by the overall cost of each platform, which is given in Table 7.6.

Table 7.6: Performance per dollar spent for each technology

Platform	Performance (MCUPS) per $ spent	Normalized Performance per $ spent
FPGA	0.34	4.9(0.34/0.07)
GPU	0.14	2(0.14/0.07)
GPP	0.07	1(0.07/0.07)

This shows the GPU to be twice as economic a solution as the GPP, while the FPGA is ahead for this particular application. Generalizing to other applications puts GPUs at an advantage because of its standard hardware and relatively simple programming model.

We have also measured the power consumed by each implementation as shown in Table 7.7. We used a power meter connected between the power socket and the machine under test for this purpose. We noted the power meter reading, at steady state, when the Smith-Waterman algorithm was running. This includes two parts: an idle power component and a dynamic power component. The idle power component can be obtained from the power meter when no Smith-Waterman algorithm implementation was running. The dynamic power consumption is thus obtained by deducting the idle power reading from the steady state power reading. We use the dynamic power figures for the FPGA and GPU implementation as nearly all of the processing

is done on the accelerator, with the host only sending query data and collecting results from the accelerator. As such, the cost and power consumption of the host could be made as small as needed without affecting the overall solution performance. The GPP implementation's steady state power figure however is used, instead of the dynamic power, as there is no distinction between host and accelerator in this case. By multiplying the power figure with the execution time (query length of 256), we obtain the energy consumed by each implementation as shown in the same table.

Table 7.7: Power and energy consumption of the Smith-Waterman algorithm implementation on all three technologies

Platform	Power (Watt)	Energy (Joule)	Normalized Energy Consumption
FPGA (clocked at 80MHz)	39	73	0.0024(73/29680)
GPU	110	3300	0.11(3300/29680)
GPP	70	29680	1(29680/29680)

This shows the FPGA solution to be three orders of magnitude more energy efficient than GPP, while the GPU solution came second with one order of magnitude energy efficiency compared to GPP. The performance per watt can thus be calculated by dividing the MCUPS for aligning query sequence of length 256 the overall power consumption in Table 7.7 for each platform. The result is presented in Table 7.8.

Table 7.8: Cost of purchase and development on all three platforms

Platform	Performance (MCUPS) per Watt	Normalized Performance per Watt
FPGA	497	414(497/1.2)
GPU	11	9.2(11/1.2)
GPP	1.2	1(1.2/1.2)

Here again, GPUs occupy the middle ground between GPPs and FPGAs. It is important however to note at this stage that the above results are very

sensitive to the technology used and level of effort spent on the implementation. For instance, if we consider the GPP and GPU implementations reported in [3] and [4] respectively, and assuming that development times and power consumption figures were similar to the GPP and GPU implementations reported in this chapter, then the resulting performance per $ and performance per watt figures of the GPP and GPU implementations would have been as shown in Table 7.9.

Table 7.9: Performance per $ and per watt for each technology using Farrar's GPP implementation and Dohi's GPU implementation

Platform	Performance (MCUPS) per $	Performance (MCUPS) per watt
FPGA	0.34	508
GPU	1.27	196
GPP	1.18	13.7

From the table, we find that the performance per $ of GPU and GPP outperforms FPGA. Therefore, no single computing technology is superior to all other technologies on all accounts. Some algorithms would be more efficient to run on GPU e.g. matrix algebra, while other algorithms would be more efficient to on GPP e.g. front end data management and control. Therefore, a heterogeneous system might be the solution to increasingly conflicting requirements, where some tasks of an application would be implemented on GPUs, others on GPPs, and other tasks on FPGAs for instance.

7.5 Conclusions

In this chapter, we presented an evaluation of the overall work presented in this thesis. We focused on a discussion of different GPU solutions presented in previous chapters as well as lessons learnt from this work in optimizing GPU programs, and a comparative study of GPUs in relation to other computer technologies. Overall, considerable performance gains can be obtained from careful consideration of thread, block and memory allocation, as well as efficient access to global and shared memory. Finally, a comparative study of GPUs, FPGAs and GPPs in the context of the Smith-Waterman algorithm implementation showed GPUs to be an economic platform for such applications, especially given FPGAs' lack of hardware standards and relatively low level programming model.

7.6 References

[1] Benkrid K., Akoglu A., Ling C., Song Y., Tian X. and Liu Y.: High Performance Biological Pairwise Sequence Alignment: FPGA vs. GPU vs. Cell BE vs. GPP, International Journal of Reconfigurable Computing, accepted in Feb, 2012.

[2] Benkrid K., Liu Y. and Benkrid A.: A Highly Parameterised and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment, IEEE Transactions on Very Large Scale Integration (VLSI Systems), Vol. 17, Issue 4, pp. 561-570, April 2009.

[3] Farrar M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations, Bioinformatics 23, pp. 156-161 (2007).

[4] Dohi K, Benkrid K., Ling C., Hamada T., Shibata Y.: Highly efficient mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs, 2010 Application-specific Systems Architectures and Processors (ASAP), pp. 29 – 36, 7-9 July 2010.

Chapter 8
Summary and Conclusions

8.1 Introduction

In this thesis, we proposed the use of off-the-shelf computing technology, in the form of Graphics Processing Unit (GPU), as a relatively low cost, high performance and programmable general-purpose computing platform in order to cope with the sheer immensity of data sets in Bioinformatics and Computational Biology (BCB) applications. The aim of this research was to develop multi-threaded design and implementations based on the architecture of CUDA-compatible GPUs for biological sequence analysis and phylogenetic analysis, assess the efficacy and efficiency of these implementations. Moreover, this thesis presented a general evaluation of the designs and implementations achieved in this work as a step towards the evaluation of GPU technology in BCB computing, in the context of other computer technologies including General-purpose Processors (GPPs) and Field Programmable Gate Arrays (FPGAs) technology.

This final chapter will first summarize the work presented in each of the previous chapters and then draw together general conclusions with ideas for potential future research.

8.2 Thesis Summary

Graphic cards have come into existence with image and graphics processing in mind as the name suggests, but newer more unified and programmable graphics hardware have been developed for more general purpose computing. These applications have more general mathematical calculations, along with GPU's unified computation characteristic, providing efficient and fast computing for non-graphics applications. Chapter 2 presented essential background of parallel computing technology and fundamentals and

characteristics of GPUs. Furthermore, the Compute Unified Device Architecture (CUDA)-compatible GPU architecture was presented including memory models and their use on CUDA-compatible GPUs with compute capability 1.0.

Aligning a query sequence to subject sequences from a large biological sequence database to find their similarities is a widely used and fundamental operation in BCB. However, biological sequence alignment is also a computationally intensive operation, and commercial desktop computers alone cannot be relied upon to perform this operation within acceptable time periods as biological sequence databases are growing at an exponential rate. To overcome this problem, chapter 3 presented a GPU-based multi-threaded design and implementation of the widely used sequence alignment algorithm, namely the Smith-Waterman algorithm, by using two task parallelization strategies. Subsequently, implementation results were presented and then evaluated comparatively with some previous GPU-based implementations and an equivalent software implementation - SSEARCH running on a desktop computer. The proposed method outperforms an equivalent CPU-based implementation by up to 15x. Moreover, we performed a comparison between two main parallelization techniques, namely inter-task parallelization and intra-task parallelization, which showed a trade-off between the two strategies in that intra-task parallelization works better for relatively small databases with smaller query sequences and inter-task parallelization performs better for relatively large databases with longer query sequences.

The Smith-Waterman algorithm adopts exhaustive search strategy which is computationally intensive and expensive. To overcome the shortages caused by the complexity of exhaustive dynamic programming algorithms, heuristic methods are developed to speed up the process of finding a satisfactory solution at a lower computational cost. Chapter 4 presented another widely used heuristic sequence alignment algorithm, namely BLAST, and proposed a GPU-based multi-threaded design and implementation of it. Subsequently,

implementation results were presented and then evaluated comparatively with an equivalent software implementation, running on a desktop computer. The final GPU implementation of our design results in a 1.7x-2.7x speed-up compared to the most optimized CPU-based implementation, namely NCBI-BLAST. While this speed-up is smaller compared to the speed-up achieved by GPU implementations of the Smith-Waterman algorithm, it is not negligible especially given the relatively low cost of GPUs. To our knowledge, this is the first GPU-based implementation of the Gapped BLAST algorithm ever reported in the literature.

A Multiple Sequence Alignment (MSA) is used to study the relationships between sets of biological sequences. It is the hierarchical extension of pairwise alignment which aims to illustrate the homology or dissimilarity of biological sequences. However, A MSA is also a prohibitively computationally expensive application as the problem of MSA evaluation can be even NP-hard. Hence, chapter 5 introduced a heuristic approach, namely Neighbour-Joining (NJ) method, in order to make MSAs computation tractable, which restricts the solution to the neighbourhood of only two closest sequences at a time, and carries on the computation between the profile and another sequence until reaching the root of the tree topology produced by a distance matrix. After that, an improved strategy to exploit the intra-task parallelization approach in the implementation of multiple sequence alignments on CUDA-compatible GPUs was presented. Subsequently, implementation results were presented and then evaluated comparatively with previous GPU-based implementations and an equivalent software implementation – ClustalW, running on a desktop computer. The proposed GPU implementation outperforms the optimised CPU-only implementation by factors ranging from 6.2x to 19.6x. Moreover, the best reported speedup in the proposed approach doubled the performance over a previous implementation of MSAs even on an older type of GPU.

Phylogenetic analysis is used to investigate the evolutionary relationships among groups of organisms or among a family of related nucleic acid or

protein sequences based upon their similarities and differences. They are helpful in inferring the history of organism lineages as they evolve over time. However, the number of possible phylogenetic trees grows in a factorial way with the number of species under analysis so that the identification of the optimal tree is computationally prohibitive. Hence, chapter 6 presents a GPU-based multi-threaded design and implementation of the Maximum Likelihood (ML) method for phylogenetic analysis on a set of aligned amino acid sequences. Subsequently, implementation results were presented and then evaluated comparatively with an equivalent software implementation – MrBayes, running on a desktop computer. The proposed method achieves 5x – 8x speed-up factors compared with optimized GPP-based phylogenetic analysis.

In chapter 7, we presented an evaluation of the overall work presented in this thesis. We focused on a discussion of different GPU solutions presented in previous chapters as well as lessons learnt from this work in optimizing GPU programs, and a comparative study of GPUs in relation to other computer technologies. Overall, considerable performance gains can be obtained from careful consideration of thread, block and memory allocation, as well as efficient access to global and shared memory. Finally, a comparative study of GPUs, FPGAs and GPPs in the context of the Smith-Waterman algorithm implementation showed GPUs to be an economic platform for such applications, especially given FPGAs' lack of hardware standards and relatively low level programming model.

8.3 Conclusions and Future Work

Following on the above, the following general conclusions can be made with the original objectives of the thesis in mind:

- Bio-sequence alignment by the Smith-Waterman algorithm: the results shown in chapter 3 shows that our proposed GPU-based method outperforms an equivalent CPU-based implementation by up to 15x. Moreover, we performed a comparison between two parallelization techniques on the implementations which shows a trade-off between the two strategies in that intra-task parallelization works better for relatively small databases with smaller query sequences and inter-task parallelization performs better for relatively large databases with longer query sequences.

- Bio-sequence alignment by the heuristic BLAST algorithm: the results shown in chapter 4 shows that our proposed GPU-based method of the design and implementation of BLAST algorithm results in a 1.7x-2.7x speed-up compared to the CPU-based implementation. To our knowledge, this is the first GPU-based implementation of the Gapped BLAST algorithm ever reported in the literature.

- Multiple sequences alignments: the results shown in chapter 5 shows that our proposed GPU-based method for the design and implementation of Multiple Sequences Alignments (MSAs) results in a speed-up over an optimised CPU-only implementation by factors ranging from 6.2x to 19.6x. Moreover, the best reported speedup in the proposed approach doubled the performance over a previous implementation of MSAs even on an older type of GPU.

- Phylogenetic Analysis: the results shown in chapter 6 show that our proposed GPU-based method for high performance phylogenetic analysis achieves 5x – 8x speed-up compared with optimized GPP-based phylogenetic analysis.

- Evaluation of GPUs in high performance BCB: the results shown in chapter 7 show the optimizations which should be applied to GPU programming for higher efficiency. Moreover, an evaluation of GPU technology in high performance BCB, using the Smith-Waterman algorithm as a case-study, compared with FPGAs and GPPs, was presented. This showed GPUs to be an economic platform for high performance BCB.

In light of the above, we can say that the original objectives of this research have been met. The following presents potential plans for future work:

- Use of GPU technology to design and implement other widely used BCB algorithms, e.g. molecular dynamics simulation, and Hidden Markov Model (HMM) applications.

- NVIDIA's next generation CUDA compute architecture, namely Fermi architecture, extended the performance and functionality of G80. Fermi improves double precision performance, as some GPU computing applications need more double precision performance. In addition, Fermi increases the size of shared memory on each Steam Multiprocessor (SM) and affords true cache architecture to applications. Porting the work presented in this thesis to Fermi GPUs would constitute a good appraisal of GPU technology evolution.

- AMD is another main GPU manufacturer. Possible future work would study GPU architectures from AMD and implement the above BCB algorithms on them to evaluate their performance compared with NVIDIA GPUs.

- As shown in Chapter 7, no single computing technology is superior to all other technologies on all accounts. Some algorithms would be more efficient to run on GPU e.g. matrix algebra, while other algorithms would be more efficient to on GPP e.g. front end data management and control. Therefore, a heterogeneous system might be the solution to increasingly conflicting requirements, where some tasks of an application would be implemented on GPUs, others on GPPs, and other tasks on FPGAs for instance. An interesting future work would research heterogeneous computer architectures and programming.

- Since CUDA is only designed for NVIDIA's CUDA-compatible GPUs, it may result in problems when porting the CUDA code into other platforms. The Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms. An interesting future work would study this portable language for portable and heterogeneous computing.